User’s Manual for CLARAty Vision Package

1. Introduction

Congratulations on selecting CLARAty, the Coupled Layer Architecture for Robot Automomy, as the platform for your computer vision application. CLARAty provides a library of written-and-tested data structures and functions to handle common image processing and computer vision tasks. It also provides facilities to quickly build your application for many targets, such as Linux, Solaris, and VxWorks, and to quickly integrate your code with existing rover code so that you can quickly springboard from algorithm concept to field-tested prototype.

CLARAty is intended to provide one, centralized, easy to access code repository. Ffuture applications can then be built on a common code base, and added back into the repository, to facilitate adapting/integrating code from separate projects, thus accelerating development in future projects. There are several other computer vision libraries just within JPL Machine Vision community, but CLARAty integrates its vision libraries with robotics libraries, so that your vision code can easily transition to testing on rover vehicles.

The CLARAty vision package attempts to provide functionality comparable to that of other packages, to minimize the difficulty of transitioning from existing libraries. It is also designed to grow as new projects require new functionality. If we missed something that you need, the package can be modified to add functionality, streamline/optimize modules, or even add competing algorithms for existing functions.

This manual outlines how to get started using CLARAty and covers the data structures and functions in the CLARAty Vision package. As a special bonus, it also lists constants used by the CLARAty modules and has a few possibly useful appendices. For additional information about CLARAty, including more details on usage and support for rovers, see http://claraty.jpl.nasa.gov. For more information about the vision package, perhaps including the latest version of this document, as well as another version of how to get started, visit http://claraty.jpl.nasa.gov/vision. For even more assistance, visit http://claraty.jpl.nasa.gov, navigate to Project ⇒ Team ⇒ Developers, and choose a developer to contact.

2. Getting Started with CLARty Vision

Here are the steps to follow to use CLARAty, as "condensed" from http://claraty.jpl.nasa.gov/new_site/software/how_to/index.html. If you find anything incorrect, missing, or problematically vague in this description, let me know (rmadison@jpl.nasa.gov) so I can fix it.

2.1 Step 1: Set up your Computer

The first step is to set up your computer to use CLARAty.

- **Get an AFS account.** If you can reach /afs/jpl/home/yourinitial/yourusername, then you probably already have an AFS account. Otherwise, visit http://eis.jpl.nasa.gov/fil/afs/client/, click any operating system and then click "Create an AFS account." It will ask you for your EIS password, which is probably the password you use for the timecard system.

- **Make sure your machine has AFS.** CLARAty is set up for unix/linux, so you'll want a machine that runs one of those. If you can see /afs on the machine, then it has AFS. If you can't see /afs, go to http://eis.jpl.nasa.gov/fil/afs/client/, click on the unix link, and click the various, self-explanatory links to set up a client. The telerobotics machines all (except Helios) seem to have AFS.

  CLARAty is not set up for developing under Windows, but you can do it with some extra effort. If you plan to develop on a PC, you might want AFS on your PC. Go to http://eis.jpl.nasa.gov/fil/afs/client/, click on the link for your operating system, then click "Check for AFS Availability" to see whether you already have an AFS client. Use the other links to get a client if you don't have one, and to learn how to log in and map your AFS space to a network drive.

- **Get added to the CLARAty group.** E-mail admin@robotics and ask to be added to the CLARAty group.

- **Set up your unix/linux environment.**
  - Run tcsh, not csh. Have Bill Adler (or other sysadmin) set that for you, or run it yourself each time you want to use CLARAty.
  - Run klog to log yourself in to AFS. CLARAty code and compiler helpers are on AFS.
  - Source /afs/jpl/group/claraty/YaM-Configs/SOURCEME-CLARAty-jpl to set up variables that the YaM make-system uses. If you also work on another project that uses YaM, it may require a conflicting script, so you probably shouldn't source the script in your .tcshrc. Instead, define aliases to the scripts, and interactively decide which to call.

2.2 Step 2: Check out some CLARAty code

The reason to use CLARAty is to have access to pre-written, pre-tested code. Here is how you get that code.

- **Make a sandbox.** This is where you can work on copies of repository code without impacting anybody else. Use the command, "yam setup –nobuild –nolink –d <dir>", where <dir> is a not-yet-existing directory that will become your sandbox. Yam will ask some questions and then pop up an editor with the file YAM.config. Quit out of the editor. CLARAty will copy files into your sandbox.

- **See what modules you want.** This document (later chapters) names modules as it describes functionality, but it is probably not up to date. For a full list of modules, visit http://claraty.jpl.nasa.gov/new_site, click on the software tab, then the package tab, and then the link for the "module database." That database is searchable by partial module name, and most module names contain obvious keywords. The database requires a password (the website says how to get it). Until you get the password, you might search through the repository on the telerobotics machines, at /afs/jpl/group/claraty/private/YaM-CLARAty/Module-Releases.

- **Check out module(s).** You can check out a module as a *work module* (which you can modify) or a *link module* (which you can look at and link with but not modify).

    - Tell CLARAty what modules you want by adding module names to your sandbox's YAM.config file. The easy way to add a module, and any modules it uses, is to cd into your sandbox and call, "yam config –add <*modulename*> [-towork]". The optional –towork flag makes the module a work module. The command will open YAM.config in an editor. YAM.config has sections that list work modules, link modules, and branch path. It should have the module(s) you just added and the SiteDefs module. If a module is listed as a work module and a link module, you might eliminate it from the list of link modules. Otherwise, just quit the editor to complete the command. You can add multiple modules this way.
    - Check out new modules in the YAM.config file by calling "yam rebuild –nobuild -nolink." If you add more modules later, call yam rebuild again. If you forget the –nobuild or –nolink flags, the process will probably hang.
    - You can edit YAM.config by hand, to change the version of a module or change whether it is a link or work module. If you add/change a link module, call "yam relink" to update the links that point to the old link module. If you add a work module, call yam rebuild to get the new code. If you change a work module, delete the module's directory in sandbox/src and then call yam rebuild to get the new code. You may need to yam relink again as well.

2.2a Optional Step: Porting to a PC

If you want to develop on a PC, your best bet is to have your sandbox in your unix AFS directory and to mount that directory as a drive on your PC. See above for how to get AFS on your PC. You can create a Visual C++ project and add the code from your sandbox directory. Alternately, if you don't have/want AFS, SSH is a convenient way to copy source code from your sandbox to your PC. You probably only need *.cc, *.h, and Makefile. It is a good idea to keep code from different modules in separate directories, as it is in the sandbox, in case you want to reinsert the code into CLARAty at some point.

Having done these things, you must now handle the following complications.

- Visual C++ version 6.0 cannot understand ".cc" as a C++ file extension. You must rename C++ files to ".cpp" before adding them to your project. I recommend writing little shell scripts, such as, "foreach i (`ls *.cc`); mv $i ${i:r}.cpp; end" to convert files between .cpp and .cc.

- Visual C++ does not use the module's Makefile. You'll want to make a project and add the code files that the Makefile lists as lib sources. If the Makefile specifies extra include directories, using a CPLUSPLUS_INCLUDES statement, add those extra include directories to your project file. If the directory paths include environment variables such as WIND_BASE (for vxworks files), you can probably printenv to see what they evaluate to, but you may not have equivalent files on your PC.

- Many CLARAty files #include headers from the sandbox's *include* directory, often accessing its subdirectories in the process. If you mount AFS on the PC, just add the sandbox' include directory to your C++ project's list of include directories. If you copy files to the PC instead, your best option probably is to copy the sandbox include directory and add it to the project's list of include directories. You will have two copies of each include file, which you need to keep synchronized, but you won't have to change a lot of #include statements to make things compile.

- Not all CLARAty code is Win32 compliant. The code uses #ifdefs to handle different operating systems, but WIN32 is not supported, so the #ifdefs might not be set up properly. Interestingly, there are #ifdefs that check WIN32 and/or MSCVER, which look like windows support, but they are not always sufficient or even correct. If you find such problems, please tell me.

- Iostream and Iostream.h are different under Windows, but not under unix. CLARAty uses them interchangeably. If you find that stream I/O items such as << and cout mysteriously fail to link in your code, try converting all of your iostream.h includes to iostream.

2.3 Step 3: Compile CLARAty Code

To compile a new or modified sandbox, call "gmake all" from the sandbox directory. This copies source code for work modules into the sandbox's *src* directory, creates links to source code for link modules in the *src-links* directory, creates links to headers and libraries in the *include* and *lib* directories, and compiles libraries and executables for all work modules. You can make specific pieces by calling "gmake bins", "gmake links", etc. Call "gmake help" in your sandbox directory for details. To recompile just one work module, you can use "ymk all", "ymk bins", etc., from inside a *src/<modulename>* directory.

By default, compiling makes libraries and executables for the architecture and OS on which you are compiling. However, you can cross compile for any target (combo of architecture and OS) named in your sandbox' *lib* directory. For a non-rover target, you can cross-compile by specifying "YAM_TARGET = <targetname>" in gmake or ymk or by calling "setenv YAM_TARGET <targetname>" at the command line.

To cross compile for a rover, issue a rover command (such as "rocky8" or "fido") at the command line before compiling. Each command defines the proper YAM_TARGET, cross compiler, and Tornado environment variables for the named rover. Some rover-related modules use the Tornado variables in the Makefiles, so compiling will fail (without a good explanation) if you have not called a rover command. I do not know how to tell CLARAty to revert to compiling for your local system after you have called a rover command, but you could try undefining YAM_TARGET. Sourcing SOURCEME-CLARAty-jpl at setup creates the rover commands. Call "alias | grep YAM_TARGET" for a list of these commands.

2.4 Adding Code to CLARAty

If you develop in unix/linux, or if you port code back from a PC, you will want to create your own CLARAty module. Here are the steps to follow.

- **Choose a module name**. While developing code, a good module name is *user_<your-telerobotics-username>*. This name indicates work in progress and makes it easy to recognize the module's author. If you have finished developing and want to return your code to the repository, choose a more descriptive name. For example, if your module detects water, call it *water_detector*. People will find your module in the module database by entering key words, so choose the most obvious key words for the name.

  If you make a module with a lame name, or you make *user_something* that is not your actual user name, somebody will send mail asking you to rename the module. You can't rename it or delete it yourself, so you have to track down one of only a couple people who can, and they are always busy, so you'll have this action item hanging over you until you die. So just don't go there.

- **Create a module.** Once you have a name, call “yam mkmodule <*modulename*>,” from any directory, to create a module in the CLARAty repository. The command brings up a Readme file where you record what the module does and what other modules it depends on. You can edit the Readme later in case you don’t know the dependency modules off hand. After quitting the editor, you’ll probably want to check the module out into your sandbox, as described in section 2.2, probably as a work module.

- **Add code to your module.** A newly created module has sample.h, sample.cc, and test_sample.cc. Replace these with your own code, but keep the general format: *.cc for library functions, test_*.cc to test those functions and provide usage examples, and *.h to export the functions and data structures. When you add your own code, modify appropriate lines in the module’s Makefile.yam to show: which *.cc files compile into bins and which into the module’s single lib file; what other modules’ libs must link into yours; and which header files need to be copied into the sandbox’ include directory. A good rule of thumb is to copy any such lines from the Makefile.yams of modules that you grabbed using, “yam config –add.”

- **Compile your code.** See section 2.3.

- **Check your module into the repository.** Being a world class programmer, you are happy to provide your code to other, less capable vision researchers by placing it in the repository. To do it right, you compile and test the code on all targets supported by CLARAty, use CVS to merge your changes with other versions that other people were working on at the same time, potentially do something to restrict access to JPL, and then release the module. I’ve not done this yet, the procedure is probably going to change soon, and in any case I think it qualifies as beyond scope for an introduction to CLARAty. If you want more information, visit http://claraty.jpl.nasa.gov/new_site/software/how_to/index.html. Or, if you want/need to release something, ask a developer for help (see section 1.)

3. CLARAty Vision Data Structures

CLARAty provides standard vision-related data structures. These are almost all in the form of C++ classes, typically providing getters, setters, I/O, and arithmetic. The larger structures, such as images, may not include large operators such as file I/O or convolution – these would appear in section 4. I have somewhat arbitrarily divided CLARAty vision-related data structures into these groups: linear algebra, frames, cameras, images, parameterized shapes, 2D grids, miscellany, and non-vision structures that you might want to know about. The following sections give the classes relevant to each group. Within the sections, each bullet gives a class name, the module and file where the class is defined, and an overview of the data and functions in the class. For a quick overview, just skim through and look at the boldface titles. For more detailed API documentation, see http://claraty.jpl.nasa.gov/vision/auto/docs/html_vision. If that documentation is out of date, contact maxb@telerobotics.jpl.nasa.gov.

Here are a few additional notes, included for completeness. You should probably skip them and jump to the next section.

- Many modules define member io() or global io_object(), often in a file <*classname*>_io.h. These use a mechanism called FDM (flexible data marshalling) to do data structure I/O. The mechanism is defined in the non-computer-vision, Data_io module, which is documented in an appendix. You can pretty safely ignore it, as I generally do in the explanations below.

- Data structure modules often contain template files, which instantiate the data structures with various template types. I remember that this is necessary because sometimes C++ forgets to instantiate some templated types that your code uses. However, I instantiate my own template if an instance is missing, so I do not use the template files, and I have not documented them.

- I refer to sub-sampling methods as *averaging* or *decimating*, translating roughly to bilinear or nearest neighbor interpolation, respectively. This is an improper use of the word *decimation*, but I use it (under protest) because it seems to be the most commonly accepted term.

3.1 Linear Algebra

Linear algebra is about points, vectors and matrices. CLARAty has a number of specialized versions of each. Most take a template parameter for the data type of their elements. Most have functions to print their data to an output stream, though I do not repeat that in each blurb.

- **Point** (points – point_t.h) is a 3-vector, useful for storing a point or a ray. Constructors copy another point or 0-3 scalars (x,y,z). Setters set x, y, z, individually or all at once, or x,y at once. Can access elements as x,y,z or as vector entries. Can multiply or divide by a scalar. Can add, subtract, dot, cross, find-scalar-distance-to, or check-equality-with another point. Can get magnitude. Getters/setters can convert between spherical coordinates (azimuth, elevation, radius.) **Dpoint** is just a *Point<double>* and **FPoint** (util_open_gl – gl_object.h) is a *Point<float>*.

- **Point_2D** (points – point_2d_t.h) is a 2-vector, suitable for image coordinates. Can construct with x or x,y. Can get x or y, or get by array index. Can add or subtract another point and multiply or divide by a scalar. Can test equality of two points. Can compute vector length or length squared, normalize the vector, or compute a unit vector. Given a second point, can compute distance, distance squared, dot product, or a point a given fraction of the way to the second point. Can compute angle (about origin) from x axis or a second point, or rotate the vector by an angle about the origin or a second point. Can compute distance to a ray-from-the-origin or to the nearest point on a line segment. **Dpoint_2D** and **IPoint_2D** are just *Point_2D<double>* and *Point_2D<int>*. **Grid_map_point** (map_grid – grid_map_point.h) is a *Point_2D*.

- **Array_2D** (arrays – arrays.h) is a 2D array, recording width, height, and a list of elements stored in row-major order. Elements may be a non-mathematical type. The array can be a sub-array in a wider memory block. To support sub-arrays, the array also records width, height, and first element pointer of the parent array, a bool saying whether the array is a sub-array, and a counter to track how many arrays and sub-arrays depend on the same memory block. To support fast indexing, *Array_2D* records a 1D array of pointers to the first element of each row of data. Construct to 0-by-0 size, or pass dimensions (and optional scalar, pointer, or iterator from which to copy data), an *Array_2d* to copy, a parent *array_2d* and sub-array coordinates. Fill by assigning another array or a scalar, accessing individual elements with operator(r,c) or an *IPoint_2D*, or using a function to fill diagonal elements of a square matrix from a pointer or iterator. Getters provide array or parent dimensions, a row pointer, a copy of a row or column, or an iterator to the first or last array elements or to any column. Additional functions transpose, resize, check equality with another array, or write to a stream. Functions for sub-arrays move or resize the sub-array within the parent (without changing data), copy the sub-array out of the parent, or check whether an array is a sub-array. **DArray_2D** is just an *Array_2D<double>*.

My tests suggest that array iterators (arrays – array_iterators.h), which are the most C++ way to index through arrays, are also very slow. If you need to step through an array quickly, it seems better to make a child class that uses protected Array_2D functions to either access the beginning-of-memory pointer or access an element with single, autoincrementing index.

- **Array_1D** (arrays – arrays.h) is a child of *Array_2D* with only 1 row. It inherits all functionality, but you can also index an element with only one coordinate. **DArray_1D** is just an *Array_1D<double>*. **Arraysum()** (analysis_edge – sky.cc) returns sum of elements of Array_1D. **Avevar()** (analysis_edge – sky.cc) is an inefficient mean-and-variance calculator for an Array_1D.

- **Matrix** (matrices – matrix_t.h) is a child of *Array_2D* that assumes a mathematical element type and defines math functions. Can construct from no params (0-by-0 size), dimensions (copy data from optional pointer, iterator, column vectors, or scalar), another *matrix* with optional sub-matrix coordinates, or one dimension and an "identity matrix" tag. Can assign another *matrix* or copy an array of column *vectors*. Can add, subtract, multiply or divide each element by either a scalar or the corresponding elements of a second *matrix*. Can transpose, transpose-and-multiply by another *matrix*, copy a column out to a *vector*, or get the minimum, maximum, or average of matrix elements. Can return a matrix with the absolute value of each element in the original *matrix*. Can convert a *matrix* of any type to a *matrix<float>*. Can concatenate two matrices side-by-side or top-to-bottom. A **DMatrix** is just a *Matrix<double>*.

  Several non-member matrix functions are defined. **Cholesky decomposition and back-substitution** (matrices – matrix_cholesky.h) solve AX=B for X, for symmetric, positive definite A. **Inverse** (matrices – matrix_inverse.h) uses numerical recipes LU decomposition and back-substitution for the general matrix, or closed form for a 2x2 *Matrix_NxM*. Output has element type M_FLOAT, which is defined as double in matrix_t.h. **SVD** and **pseudoinverse** (matrices – matrix_svd.h) seem to be from Numerical Recipes. Matrix operators in (matrices – matrix_operators.h) add, subtract, multiply, or divide a matrix by a *double* or a scalar of the same data type as the *matrix*, modifying the *matrix* in place.

- **Vector** (matrices – matrix_t.h) is a child of *Matrix* with only 1 row. It generally inherits all *matrix* functionality. Can typecast a 1D to 3D *Vector* to a *Point* or construct a 3-vector by passing a *Point*. Can assign, resize, dot or cross with another *vector*, access elements using array indexing, and get magnitude, minimum value, or maximum value. Additional, non-member functions (matrices – matrix_operators.h) let you dot, cross, get magnitude, get sum of elements. **DVector** is just a *Vector<double>*.

- **Matrix_NxM** (matrices – matrix_nxm.h) is a child of *Matrix* with statically allocated data space and dimensions N and M given by template parameters. It cannot be a sub-matrix. Presumably it is faster than a regular Matrix. Construct with no params or copy from column vectors, scalar, data pointer, iterator, *Matrix*, *Matrix_NxM*, or *Array_2D*. Can assign from another *Matrix_NxM*. Can access elements by (r,c) coords. Can resize, calculate transpose, or transpose in place.

- **RMatrix** (matrices – rotation_matrix.h) is a 3x3 rotation matrix streamlined to take advantage of the size and orthogonality. It inherits a 3x3 array from base class **Base_RMatrix** (same file, used only here.) Construct from another *RMatrix*, a *Base_RMatrix*, a 3x3 *Matrix*, three column *Vectors*, 1-3 rotation angles, or no parameters (zero matrix – not a valid rotation matrix). If you have row vectors, construct as if they were column vectors and then transpose. Can copy from static Identity or Zero matrices. Can access elements with (r,c) coords. Can transpose or invert (same operation), subtract another *RMatrix*, multiply two *RMatrices* or multiply an *RMatrix* onto a *Point*. Can cast to *Matrix* or *Matrix_NxM*. Can extract roll, pitch, and yaw angles.

- **UnitVector** (matrices – unit_vector_t.h) is an array of 3 statically allocated elements. It is probably faster than a regular vector. Construct from no params (0,0,0) or construct/copy from three values or a Vector (the values are normalized before assignment). Can typecast to a *Vector* or a *Point*. Can add, subtract, dot, or cross two *UnitVectors*. Can access using array index or as x,y,z. This class is probably obsolete. It appears to be a primitive version of Point.

- **Htrans** (transforms – htrans_t.h) is a homogeneous 3d transform: a rotation matrix plus translation vector. Constructor, setters, and getters can use rotation matrix, fixed angles, or "DH parameters". Can get transform or translation to another *HTrans*. Can invert transform. Can premultiply onto another transform, a 3d point, a roll angle, etc. **Location** (transforms – htrans_t.h) is just an *HTrans<double>*.

- **Quaternion** (transforms – quaternion_t.h). Construct from axis/angle, a *matrix*, 4 scalars, a quaternion, or 3 angles. Convert to/from 3 angles, from axis/angle, to angle, to/from rotation matrix. Interpolate toward another quaternion. Negate, conjugate, raise to a power, scale by scalar, sign flip so last element is positive, normalize to unit quaternion, get norm, quaternion multiply, divide, =, ==.

- **Float_matrix** (jplpic – FloatMatrix.h) and (camera_model_jpl – float_matrix.h) are redundant with each other and the other CLARAty matrices. Its functionality should be assimilated and its references in the JPLPic code replaced.

- **More Matrices and vectors** (jplpic – Mat3.h) and (camera_model_jpl – img_mat3_pub.h). These are matrix and vector routines specialized for 3x3 matrices. The two sets of code are redundant. Consider a specialized matrix class similar to rotation matrix, just for 3x3 matrices.

3.2 Frames (a.k.a Kinematic Chains)

Robotics applications involving articulated arms may assign a reference frame to each link in the arm and record the coordinate transforms between adjacent links. Adjacent transforms can be concatenated to create a coordinate transform between any pair of reference frames on the arm. CLARAty stores these "kinematic chains" of transforms using the *Frame* class. The Frame class appears in various parts of the vision code, so even if you expect to do pure vision, you should still learn about Frames.

Continuing the example, a *Frame* represents the coordinate system of one link. It has a pointer to an adjacent, "parent" link and a coordinate transform to that link. It also has pointers to "child" and "sibling" links, so that you can build a kinematic tree, not just a chain. It provides functions to quickly convert coordinates between any pair of *Frames* in the tree. The Frames of a tree can be collected and stored in a *Framestore*, which can lock the entire tree as you modify one node, to be safe in a multithreaded environment. When using *Framestores*, you do not access the *Frames* directly. Instead, you extract a *Frame_l*, which records a *Frame* and its associated *Framestore*.

- **Frame** (frame – frame.h) is one node in a kinematic tree. It has a name string, pointers to parent, first child, and next sibling *Frames*, and a *Location* (homogeneous transform relative to a parent.) Getters access all of these except the *Location*, which is public. Construct with a name, optional parent (else create a root) and optional *Location* (else identity.) Destructing a *Frame* reparents its children. Member function location_of() extracts the transform to any other *Frame* in the tree and optionally applies it to a parameter *Location*. Additional functions: copy from a *Location* or *Frame*; get a common ancestor with another *Frame*; see if a parameter *Frame* is an ancestor; set the *Location* with respect to parent or a parameter *Frame*; reparent to a *Frame* in same tree; see if the *Frame* is a root (perhaps unsafe); and print self or children in *Parse_Block* format. **Frame_h** (frame – frame.h) is a pointer to a *Frame* plus a complete set of pass-thru functions with default return values in case the pointer is unassigned. Construct empty or pass a *Frame* pointer. It is not clear what value this class adds.

- **Frame_l** (frame – lock_frame.h) is a pointer to a *Frame* and a pointer to the *Framestore* that contains the *Frame*. *Frame_l* has the same interface as *Frame* (using pass through functions), except for the following. Most *Frame_l*s are constructed by Framestore's member functions, but you can also construct by copying another *Frame_l*, passing a *Framestore* (no *Frame*), or passing nothing. Member functions handle *Frame_l*s, not *Frame*s, and they short circuit if any parameter *Frame_l*s are not in the same *Framestore*. You can assign a *Frame_l* from a *Location*, but not from a *Frame*. *Frame_l* has additional functions to: test whether the *Frame* is assigned; get the *Frame*'s location; compare *Frame_l*s (== and !=); return the *Framestore*; see whether another *Frame_l* shares the same *Framestore*; and do stream I/O.

- **Framepair_l** (frame – lock_frame.h) has two *Frame_l*s, to represent a primary frame "with respect to" a secondary frame. Construct with both *Frame_l*s, just the primary one, or neither (primary frame unassigned.) There are functions to test whether the primary *Frame* is assigned, get the transform from primary to secondary, or get the name of either *Frame* or a compound with both frames' names. The two *Frame_l*s are public, so all *Frame_l* functions apply as well. It is not immediately clear what value this class adds.

- **Frame_Data** (frame – lock_frame.h) has a *Frame* name, a parent name, a transform, and the name of the *frame* that the transform references. Construct from a *Frame_l* (no reference frame), a *Framepair_l* (frame plus reference frame), or the four fields; or assign from a *Frame_l*. It is unclear why you would want this struct except to facilitate I/O.

- **Framestore** (frame – lock_frame) collects one kinematic tree worth of *Frames*. It records its own name string and an STL Map of frame name strings to *Frame* pointers. A *Frame* in a *Framestore* can only have relatives in the same *Framestore*. A *Framestore* cannot have multiple frames with the same name. Construct a *Framestore* empty, with only a name, or from a *Parse_Block*. The *Parse_Block* has one entry with label "name" (the *Framestore* name) and other entries whose labels are non-negative integers and whose values are parse blocks that each define on *Frame*. Labels need not be in order, but a missing integer stops the parsing. Non-integer labels are ignored. *Framestore* defines functions to: create or modify a member *Frame* (from optional name, parent name, and location, or with info in a *Frame_Data*); lookup a *Frame_l* by name string, and optionally copy it into a *Frame_Data*; remove a *Frame* given its *Frame_l*; copy the *Framestore*; merge in the entries of another *Framestore*; get a list of *Frame_l* for all *Frames* in the *Framestore* or all ancestors of a parameter *Frame*; convert a list of *Framepair_l* into a list of transforms between each pair (does that really belong here?); and print the *Framestore* contents to screen or *Parse_Block*. Virtual functions to lock and unlock the *Framestore*, to make its operations threadsafe, do nothing by default. **Framestore_pthread** (frame – lock_frame_pthread.h) is a child of *Framestore* that defines the lock and unlock functions to use a pthread_mutex, making the class thread safe.

3.3 Cameras

As of this writing, a camera is implemented with three classes. Class *Camera*, which your code would actually handle, is a wrapper (pointer + pass-through functions) for a *Camera_impl*. Class *Camera_impl* contains a list of standard camera parameters, virtual setters and getters, and image-acquire functions. One class *Camera_impl_<type>* for each specific type of camera inherits *Camera_impl* and overwrites functions, for instance to initialize the list of camera parameters. These classes try to add camera-specific functions only within the existing *camera_impl* interface. A parallel set of classes (*Multi_camera*, *Multi_camera_impl*, and *Multi_camera_impl_**) describe array of cameras that could conceivably acquire images synchronously.

Why don't we combine *Camera* (the wrapper) and *Camera_impl* (the real code)? Because potential descendents of *Camera* (*e.g.,* VideoCamera or ColorCamera), can point to any *Camera_impl_**, but descendents of *Camera_impl* must each have their own set of descendent Camera_impl_*. The problem would go away if each *Camera_impl_** could only sensibly inherit from one descendent of *Camera*, but even in our small example here, a color video a camera could sensibly inherit from either color or video camera. Also, CLARAty classes such as motors require the same sort of separation, so it seems reasonable to retain the common pattern in the vision package.

The following bullets give a slightly more detailed view of each class. For even more details, such as a list of *camera_impl_**, see http://claraty.jpl.nasa.gov/vision/auto/docs/html_vision, and click on "acquisition".

- **Camera** (camera – camera.h) is a *camera_impl* pointer and a *camera_model* pointer. Constructor sets both, getters get either, and setter sets camera model. Many more getters and setters are pass thrus to *camera_impl*. More pass-throughs acquire 1-band or 3-band *camera_image* (and set their model).

- **Camera_Impl** (camera – camera_impl.h) is a list of features for a specific camera. Features are of class *dev_feature* (camera – dev_feature.h), which has fields such as id, value, min/max, and is_on. *Camera_impl* has virtual getters, setters, and acquires (called from class *Camera*.) Getters and setters access the list of features, while acquires are null – this is a base class. Can construct by copying another *camera_impl*.

- **Camera_Impl_*** (various modules and files) inherit *camera_impl* and redefine virtual functions. **Camera_impl_proxy** (camera – camera_impl_proxy.h), which I don't understand, is purportedly helpful in wrapping your own cameras into *camera_impls*. **Sim_camera_impl** (camera – sim_camera_impl.h) is a simulated camera that "acquires" 4x3 images of constant intensity.

- **Multi_Camera** (camera – multi_camera.h) is the same as a *Camera* except that it has a pointer to an array of *camera_models* and a pointer to a *multi_camera_impl*. Construct by passing a *multi_camera_impl* or an array of *camera_impl*s. The number of cameras is a template parameter. The getters, setters, and 1-band acquire all take a camera_num parameter, so you can access one camera at a time. There is no 3-band acquire, but you can pass an array of images and have one call acquire with all cameras. The *multi_camera_impl* decides whether the cameras are synchronized. Setters and getters (except *for multi_camera_impl* and *camera_model*) are pass-throughs to the *multi_camera_impl*.

- **Multi_Camera_Impl** (camera – multi_camera_impl.h) is an array of *camera_impls,* plus setters, getters, and acquires to access the *camera_impls*. These functions have the same names as those in the individual *camera_impls* and in class *multi_camera*. The multi-image acquire() function does not synchronize the acquisition of the *camera_impls*.

- **Multi_Impl_*** (various modules and files) are child classes of multi_camera_impl that retain the same interface but redefine the setters and getters.

- **Camera_Model** (camera_model – camera_model.h) has image height and width, two fields to identify the camera's *Frame*, and getters and setters for these fields. It defines virtual functions to convert between pixel coordinates, lens-distorted pixel coordinates, and vectors from camera pinhole through pixel. It also defines confusing functions for file and stream io and factories.

- **Camera_Model_*** (various places) are specific camera models. They add parameters with getters and setters and redefine the virtual functions for pixel conversion, construction, and I/O. **Camera_model_matrix** (camera_model – camera_model_matrix.h) looks like a Tsai camera matrix and a 4-parameter distortion vector. It also has a subsample() command that just modifies the camera model. **Camera_model_jpl** (camera_model_jpl – camera_model_jpl.h) has a *JPLCamera* pointer and I/O functions for handling CAHV and CAHVOR files. Most of its functions are pass-throughs to the *JPLCamera*.

- **CameraModel** (camera_model_jpl – Camera.h, jpl_cmod.h.) It looks like somebody tried to split JPLCamera into two parts, along the lines of Camera_Model and Camera_Model_JPL, gave up, and did not delete the code. Camera_Model is currently a rather hollow parent of JPLCamera, and other tha that is never used in its own right. It should probably go away.

- **JPLCamera** (camera_model_jpl – Camera.h, jpl_cmod.h) stores the vectors, scalars and matrices found in a CAHVORE, CAHVOR, or CAHV file, plus the image dimensions that may or may not appear in such a file. These parameters constitute a camera model. *JPLCamera* has functions to: test for near equivalence with another model; scale/rotate/translate the model; rotate the model to parameter axes; and rotate/translate the model and another "rigidly attached" *JPLCamera* to put the local model at a specified pose and maintain the relationship between the two cameras. *JPLCamera* has I/O functions to:

read/write files; read from another *JPLCamera*, a memory block, or various combinations of parameters; and print the CAHV(ORE) vectors, the derived parameters, a one-line check for common errors, or the camera position and "axes" (purportedly different from A, H', V').

JPLCamera has functions to extract information from the model: camera axes; projection, rotation, and translation matrices; horizontal, vertical, and diagonal FOVs; and 2D-to-3D or 3D-to-2D projection matrices. It has functions that use the model to: find the 3D ray through a pixel or the pixel intercepted by a ray, find the angle between rays through two pixels, and find a projection matrix between two CAHV cameras, which might mean the fundamental matrix. Converting between rays and pixels is done by *jpl_cmod_cahv*()*, described below.

JPLCamera has at least four functions to rectify images from CAHVORE (and sometimes CAHV) to CAHV, generally creating and using a lookup table to map warped to unwarped coordinates. Nothing appears rectify from CAHVOR, except one function that uses a rectification table that cannot be created because the table generator function is #if 0'd out. Member variable rectification_tables is a **NamedCache** (camera_model_jpl – NamedCaches.h), meaning that it has some chance of persisting in memory while you are not using it. A member function finds the old table or allocates new memory that can be refilled using *st_warp*()* functions. A second member variable, rectificationTable, appears to be vestigial. Locally defined struct **RectMapKey** (camera_model_jpl – Camera.h) contains header information for the warped and unwarped. Global methods (should be members) test if 2 *RectMapKeys* are mostly equal and write one's contents as a string. The struct is only used to assemble a name for rectification_tables, which is why it appears here rather than with the st_warp*() functions.

JPLCamera uses a *MemoryManager* to safely handle memory.

CameraModel and *JPLCamera* functions return NavErr, which is a little silly because nothing in the module defines NavErr. Presumably it is defined in a client application. More sensible would be to have the module define its own errors, and let the client copy them. There are probably other sensible options.

- **CAHV(OR(E)) handlers: img_cmod_cahv*()** (jplpic – img_cmod_pub.h) and (jplcamera – img_cmod*.c) do various things with CAHV, CAHVOR, and CAHVORE models (abbreviated CAHV(OR(E)).) They should be part of the *JPLCamera* class, but they are legacy code, and have not been incorporated. Instead, they use separate parameters to handle the CAHV(OR(E)) vectors, the covariance matrix (S) between the vectors, derived quantities (hs,hc,vs,vc,theta), and the covariance (s_int) between them. There are functions that …

    - Read/write CAHV(OR(E)) files.
    - Calculate the derived parameters, camera orientation (as rotation matrix), or camera position plus orientation, all from the CAHV vectors.

- Use a CAHV(OR(E)) model to convert a pixel's 2D coordinates into a 3D ray, or vice versa, or find the 2D coordinates of the vanishing point of a ray. Each also can calculate the partial derivatives of the 2D/3D transform.
- Convert a pixel's 2D coordinates into a ray with one camera model and then back to 2D with a second model. There are variants to convert between two models of the same type (e.g. CAHVOR) and others to convert between CAHV and either of the other two models.
- Find the new values of CAHV(OR(E)) and S (the covariance of these vectors) if the camera's reference frame translates or rotates, the image is shifted or scaled, or the camera is reflected off a mirror plane. For the case of shifted image, the three model variants are identical. For scaled image, the variants differ only in the size of S.
- Take two similar CAHV(OR(E)) models, representing a stereo pair, and generate CAHV parameters for a rectified stereo pair (CAHV or CAHVORE) that has the same pinhole positions, about the same field of view, and a stereo baseline nearly aligned to the image horizontal axis.
- Find a CAHV model similar to an input CAHVOR(E) model but with orthonormal A,H',V', and with hc/vc/hs/vs covering the CAHVOR(E) model's FOV without distortion

The three img_cmod_cahv*.c files define static cmod_read_scanstr(), which probably ought to be declared just once.

It seems silly to pass all of the parameters to all of the functions, rather than having a CAHV object that can operate on members and perhaps be inherited to make CAHVOR or CAHVORE classes. In fact, it appears that class **ImgCMod** stores all of these parameters and is used by img_cmod_*() functions defined in img_cmod.c, which are similar to img_cmod_cahv*() but handle all three CAHV(OR(E)) models. I do not know where the class is defined or what header file declares these generic functions.

There are two sets of synonyms for these functions. They are **jpl_cmod_cahv*()** (jplpic – jpl_cmod.h) and **cmod_cahv*()** (jplpic –img_cmod_pub.h), which are #defined to img_cmod_cahv*() in said header files. I do not know why we use all of this #defining. Both headers are improperly placed in *Jplpic*. I say improperly because they relate to a camera model, not to an image format, and because jplpic does not call the functions. They are called in the camera_model_jpl module.

The two header files, in the JPLPic module, #define some constants (IMG_CMOD_MAX_FILE_BYTES, PI and FAILURE) that are unused, and another (EPSILON) that conflicts with definitions elsewhere. The headers define TRUE, FALSE, and SUCCESS, the latter being a return value that (jplpic – filename.c) expects for functions in a *FormatTagT* struct. Viscommon.h defines TRUE, FALSE, SUCCESS, and FAILURE, without all of the extra complication, and should probably replace the #inclusion of the two header files in stereo.h and image_parse.h. Then the two header files could move to camera_model_jpl, where they belong. Also, note that jplpic does not actually need stereo.c. Finally, consider replacing viscommon.h with the CLARAty equivalent.

Finally, (camera_model_jpl -- jpl_cmod_cahv.h and .cc and mat3.h) declare/define a few extra jpl_cmod_cahv* but they are used.

3.4 Images

We define the basic *Image* structure and have other structures that inherit or use it. *Image* and its children have screen I/O routines, but I do not discuss them. We also support third-party *JPLPic* and *ATImage*, which are not based on Image. Although undesirable, we support the third party formats (and provide converters) so that we can drop-in-replace code to support third party upgrades. It might be good to add a class that just has dimensions and a data pointer, which would be useful for very simple third party applications that do not want any overhead in their Image struct. It would be really good if *JPLPic's* operators could be separated out from the main data structure, as upgrades to the operators would not then affect the main structure, and we could conceivably merge the *Image* (or *Matrix* or *Array_2D*) and *JPLPic* classes.

- **Image** (image – image.h) is a *Matrix* that also supports interpolation to extract values at non-integer coordinates. Construct empty, or give dims and optional initializer (scalar, data pointer, or iterator), or another Image or Array_2D to copy, or an Image and sub-image bounds to create a sub-image in another Image. Image inherits all functions from *Matrix* and *Array_2D*, including accessing with operator(r,c) or operator(*point_2d*); moving, resizing, or extracting a sub-image; copying a *matrix*, an array column *Vectors*, or a scalar; adding, subtracting, multiplying or dividing each element by either a scalar or the corresponding elements of a second *matrix*; transposing, transpose-and-multiplying by another matrix, copying a column out to a *vector*, or getting the minimum, maximum, or average of matrix elements; converting to a *matrix<float>*; copying a row or column or getting a pointer to a row; and resizing the *image*. In addition, you can switch interpolation between BILINEAR (average of 4-neighbors weighted by their contribution to pixel area) and NOINTERP (truncate coordinates – the default). It would make sense to have nearest neighbor interpolation (add 0.5 before truncating.) A second BILINEAR option (weighted average of 4-neighbors based on Euclidean distance to pixel center) may not appear in the current version. Can access interpolated pixels using "pixel(x,y)", which is the only function where the column coordinate precedes the row coordinate. Can get a pointer to the beginning of the image, in case you need to rapidly cycle through the image elements. **GetImageWin()** (analysis_edge – sky.cc) copies a rectangle of an *Image* into an *Array_1D*.

- **RGB_Image** (image_rgb – rgb_image.h) has three *Images* and several functions to access them. Construct empty, pass dims to create empty *images*, or pass 3 *images* to copy. Function get_color(x,y) returns an **RGB_Color**, which is 3 values of the *RGB_Image*'s data type. Setters take x,y,r,g,b or x,y,*RGB_Color*. Get and set image dimensions of the *RGB_Image* rather than the bands. Can get iterators or pointers to each band. Can convert to/from an array of rgb triplets.

- **Camera_Image** (camera_image – camera_image.h) is an image that also has a *Camera_Model* pointer, a time stamp, and a frame number. Constructs like an image except that you can also pass various combinations of *camera model*, time stamp, and frame number. Has pass-through functions to the *camera_model* to attach the *camera_model* to a *framestore* or get the camera's *frame*.

- **Point_Image** (point_image – point_image.h) is a *Camera_Image* where each element is a *Point<double>*, representing a 3D point as seen by a camera. It also has a mask array (bool, to say which pixels are valid) and a confidence array (float), a reference frame (need not be the same as the camera's), another parameter dealing with frames, and a distance "infinity" giving the maximum depth of the points. Construct with optional image dimensions, reference frame, and filler value. Constructors and public function rebuild() build a *Point_Image* from a reference frame, two camera models, a horizontal disparity map, and optionally a vertical disparity map and/or corresponding mask. That disparity map is Ames-style – the *JPLStereo* disparity maps differ by several scaling factors including a cross product that differs for each pixel, so there is no ready conversion. The conversion of disparity map to 3D image belongs with camera model, not in *Point_Image*. There is also a function to find the 3D point for one pixel. Reference frames for the *point_image* and the two cameras must all be in the same *framestore* and all be related through some inheritance there, otherwise the location_of() commands in rebuild() will fail. In addition, if you use CAHVOR cameras, even though you can't use a JPL stereo disparity map, you must set the cameras' reference frame to match the CAHVOR parameters. There is a pass-through to attach the image to a *framestore*. There are functions to extract the points that satisfy a predicate, clear the mask pixels for points that fail a predicate, convert to a point cloud, "decimate" by an integer factor, or transform all points by a *Location* or *Frame_l*. The same file has **BoundingBox**, a class with one function that determines whether a *Point* lies within an axis-aligned bounding box.

- **Normal_Image** (point_image – normal_image.h) is essentially a *point_image*, except that its base data type is *Unit_Vector* rather than *Point*. It is used to represent surface normals rather than locations. It has parallel *Array_2Ds* to store residual and residual_ratio, two additional measures of confidence in the associated surface normal. Function build_from_point_image() fits a plane to a square neighborhood around each point in a *point_image* and records its normal at the corresponding point in the normal image. Mathematically, it subtracts the center point's coordinates from each neighborhood point, builds the covariance matrix of the resulting cloud, does an SVD, and uses the minimum eigenvector as the normal. This idea is sound, but I'm pretty sure you have to subtract the average of the point cloud, not the center point, to be mathematically correct. In addition to the normal at each *point_image* patch, the function records the smallest eigenvalue as the residual, the ratio of smallest eigenvalue to middle eigenvalue as the residual ratio, and the square of "1 - residual ratio" as the confidence. Points whose neighborhoods are not fully on the *point_image* or have more than half of their points with 0 as their mask value receive a 0 in the *normal_image* mask and receive no normal entry. *Normal_image* also has a fast, less robust function that calculates normals by crossing vectors between each *point_image* point and two of its neighbors. The file also has a **BoundingBox** class with a function to tell whether a unit vector is in the axis-aligned bounding box described by two member unit vectors.

- **Image_Pyramid** (image_pyramid – image_pyramid.h) is an array of *Images* and an integer dimension for the array. Each image after image[0] is a half-sized (rounded down) version of the previous image, created by convolving a smoothing filter at every other pixel of every other row in the previous image and extracting those pixels. Generate the pyramid by calling create_pyramid() or a constructor and passing image[0], an optional number of pyramid levels (default=3), and an optional smoothing filter or filter size. If you pass a filter size, create_pyramid() will skip convolution (size 1) or generate filters (sizes 2, 3, or 5.) Coordinates (x,y) in one image correspond to (2x,2y) in the next larger image if you use an odd-sized filter or (2x-½ , 2y-½) if you use an even-sized filter. For "border" pixels within half a filter-width of the input image border, convolution crops the filter to fit on the image and rescales it to the same gain as the full filter.

- **Point_cloud** (Point_cloud – point_cloud.h) is an *Array_1d<Point>* with an additional variable giving the number of points being used. The constructor sets the number of points to 0 and can specify the maximum number of points (default = 320*240). Unlike a *Point_Image*, there is no implied order to the points, and no mask array to indicate invalid points. There are functions to transform the cloud by a *Location*, delete all points, add a point (unsafe), append the points from another *point_cloud*, and change the allocated size of the array (unsafe.) It also inherits all functionality of *Array_1d*, which are mainly getters for array pointer and allocated size, and index notation to access particular members.

- **ATImage** (arc_slog_tracker – atimage.h) is an image format that packs data. Members are dimensions, data pointer (data is packed into long ints), bits per pixel, number of long ints required to store data, and number of extra pixels to reserve on each (left and right) border. Most members have getters. Construct to an empty image, or pass values for all members, or pass all but the image data, or pass another *ATImage* or an *Image*. Has separate Init functions to allocate and pack data, but may call them from the constructors. Has a separate function to allocate space for the data, so you can safely allocate after the constructor. Defines assignment, add-image, subtract-image, and multiply-by-constant operators. Has functions to convert to/from an *Image*, read/write a PGM file, write an Inventor file, and write image data bit-by-bit to screen or byte-by-byte to a file. Seems silly to duplicate the CLARAty Image format. Why do we?

- **JPLPic** (JPLPic – JPLPic.h) is the image format used for many applications in the Machine Vision group. It currently is divided among a header file and seven cc files for basic functionality, drawing, file-I/O, regular image operations, large-buffer image operations, colormap handling, and using the image as a map. Basic structure and function (mainly from JPLPic.cc) is described here. Functionality that looks more like operators and less like structure is described in other sections.

Drawing, file I/O, colormaps, and filtering are separate conceptual entities that warrant their own modules. Such modules could have a *JPLPic* pointer to tie an instance to a particular *JPLPic*, but their presence would not clutter *JPLPic* in applications that do not require them.

JPLPic records a basic image as number of rows, number of columns, and a char pointer. It accommodates different pixel data types with an enum *PixelType* of all standard data types, a member variable to record the *JPLPic*'s pixel type, and the use of **AnythingT** (JPLPic – JPLPic.h), a union of all standard data types, in member functions. There are also member functions to get bytes-per-pixel and bands-per-pixel for arbitrary pixel type or the image's pixel type, functions to convert between pixel type enum value and description string, and some macros to test things about a pixel type, such as whether it is a scalar or vector. The macros should probably be inline member functions. According to notes in the code, *JPLPic* uses cumbersome *AnythingT* instead of elegant templates because flight code cannot use templates and because some code must operate on multiple images of differing pixel types. CLARAty is not flight code, but if CLARAty is to incorporate the latest version of Machine Vision group products, and those are built as flight code, then we must work with their restrictions.

JPLPic accommodates sub-images by recording the number of bytes between the beginning of successive rows and recording whether the *JPLPic* owns its own pixel memory. There is no count to know when image data memory is no longer used. Instead, one image owns the data, which seems less robust.

JPLPic uses weak constructors that set non-allocated variables and has Init() allocate/fill the remaining members, which is safer than the way Image does it.

JPLPic supports images composed of two fields stored as consecutive blocks. A member variable tells whether the image is divided into fields. File-I/O and sliding-sum image operators take this variable into account, though I'm not sure that drawing or regular filter operators do, so they may munge the edges of the fields.

There are getters/setters for most member variables and a function to print member variables. There are getters/setters for individual pixels, including a getter that interpolates floating point coordinates on an 8-bit image. There are functions to get a pointer to the image's first pixel or the pixel at given row/col coordinates, both with options to typecast to appropriate data type pointer and to wrap around when row/col are outside image bounds. There is a function to set the *JPLPic*'s data pointer, with appropriate housekeeping. Has function to find the row/col of the pixel in an xyz-vector image that is nearest an input (x,y) point. Declares but does not define a function to find the new coordinates of a pixel if the image were to be rotated.

There are copy functions to: fill an existing *JPLPic* by copying a *JPLPic* (sub)image, a colormap entry (pic_filter.cc), or a scalar or memory block with optionally embedded rows/cols/datatype header (pic_io.cc); create a new *JPLPic* with copied header and no data; or create/modify a *JPLPic* to point to (not copy) a (sub)image in a memory block (pic_io.cc) or another *JPLPic*, create an ARGB image from separate bands stored as char * or *JPLPics*; and create two *JPLPics* pointing to the two fields of an existing image.

There are locally defined macros to check if a number is in bounds, take absolute value, choose minimum or maximum, etc., which should be replaced with cl_*().

JPLPic accommodates color-indexed images using its own colormap, which it represents with four member variables. The first two are a statically allocated array of (#defined) default colormap size and a pointer that can be allocated to a larger size. The third is a pointer to whichever of the first two is active. The fourth is a vestigial member that is always null. One file (jplpic – jplpic.h, pic_cmap.cc) has colormap handling functions that: allocate the large colormap; copy entries from another colormap; load a greyscale, color-scale, or hard-coded, rainbow color map; print the color map; set/get one colormap entry; and convert an image from colormap indexed to RGB. Another file (jplpic – cmap.h) has 5, vestigial functions for dealing with color maps. One fills params rmap[], gmap[], and bmap[] the 256 colors made from 8 values of red, 8 values of green, and 4 values of blue. Another converts red, green, and blue values (0-7,0-7,0-3) to the index of that color on the above maps. Another converts images of 8-bit red, green, and blue values to an image of index values. These first three functions assume that bits are in reversed order. The two remaining functions seem to be copies of jplpic's grey-scale and rainbow color map generators. I recommend that the colormap be separated into its own module, and attached to the drawing class and other classes that require it. I doubt that the average image requires it.

JPLPic supports using the image as a map by mapping floating-point coordinates to row/col coordinates. Member variables (with getters and setters) record scale and offset, strings naming the units for the float coordinates and bools to say whether the *JPLPic* owns (i.e. must free) the strings. There are functions (jplpic – jplpic.h, pic_units.cc) to scale the scale variables, resample the image to a new scale, convert between float and row/col coordinates, and get the address of a pixel at float coordinates as a pointer to any standard data type. The address/coordinate getters can handle maps that wrap around. There is a function to get the distance between two pixels after resolving wraparound in their coordinates. There is a big function, in 3 variants, to "annotate" an image to show the float mapping. That seems to include magnifying an the image and scaling it to 8-bit grey, drawing a grid of specified (float) period and (float) anchor point, drawing a special marker at coordinates (0.0,0.0) or optionally the wrapped equivalent, and overlaying text with the float units name, offset, and scale.

A member variable records the actual number of bytes allocated to an image, so that it need not reallocate if the image is resized to a smaller size.

There are a number of statistics functions declared in jplpic.h that appear to not be defined: ComputePixelStats, WriteStats, ExposureStats. It may be that these existed in a former version, because my older notes describe a function to find the mean, variance, and number of pixels below and above thresholds in a region of the image (could overrun end of image) and a second one to write image size, mean, and variance to a file.

JPLPic uses a *MemoryManager* member variable instead of standard new and delete functions.

JPLPic files appear in the JPLPic, gestalt_navigator, and visual_odometry_jpl modules. They may be mutually incompatible. The JPLPic files should be removed from the latter two and replaced with links to the JPLPic module.

JPLPic.h defines two macros to detect whether a 3D point or a disparity value are represent "no-data". They are not used. They do not belong in this file. And they may even fail, because the 3D point does not test against the customary no-data value, and the value for invalid disparity is defined differently across applications.

- **JPLPic_converter** (jplpic – jplpic_converter.h) has two functions to convert between a *JPLPic* and an *Image*. They are put correctly in a separate file from *JPLPic*, so that *Image* and *JPLPic* needn't know about eachother.

- **Science_Image** (Analysis_edge-R1-00a – Science_analysis.h) is an *Image<BYTE>*. Construct empty, or with dims and an optional BYTE* to fill data. Function mask() *AND*s the image with a parameter image, setting pixels to 0 where corresponding pixels of a parameter *Science_Image* are zero. There are many unused variables and declared-but-undefined functions, suggesting that the class is unfinished.

- **In_region()** (analysis_edge – layeranalysis.cc) tests whether coordinates are on an *Image*, which is probably redundant with something in *Image*.

3.5 Parameterized Shapes

Here are classes to fit data to parameterized shapes and store the parameters. They probably warrant one or more modules of their own.

- **Ellipse** (Analysis_edge-R1-00a – Science_analysis.h) stores the 5 parameters of an ellipse (x,y,a,b,theta) plus size, which is used to store ellipse area. This should probably be a class/module of its own, and the crater detection code should probably use/improve it.

- **SearchEllipseList()** (Analysis_edge – Science_analysis.h) checks whether parameter coordinates are within 1 pixel (each x and y) of any of a list of ellipses. It checks at intervals of $\pi/8$ around the ellipse. Perhaps belongs to *Ellipse*.

- **CreateEllipse()** (analysis_edge – layeranalysis.cc) fills an *Ellipse* based on a list of points. Ellipse center is the point centroid plus a parameter offset in x and y. Ellipse size is the number of points used to calculate the ellipse. Ellipse axis lengths are the square root of the eigenvalues of the covariance matrix for the points, scaled so that the area of the ellipse equals the number of points. Ellipse theta is the angle of the major axis, CCW from vertical, measured 0 to pi. Probably should be merged with crater detection code's fitter, and become class *Ellipse_Fit_Moments*, as with *Plane*.

- **Plane** (analysis_terrain_morphin – plane.h) is a plane, parameterized as z=Ax+By+C. Has function to calculate z given (x,y). Has function to calculate the roll and pitch of a robot sitting on the plane with a given yaw. Has constructors, assignment operator, and I/O operators. Has public fields for residual (chi sq plane fit) and quality (according to N.R.C.)

- **Plane_fit_moments** (analysis_terrain_morphin – plane.h) has a *Plane* and the functions and data required to fit the plane to a set of points. It records the number of points, the sums used in a standard least squares fit, and some flags. It has functions to clear the sums, add a point (with or without weight), add/subtract/assign sums of another *plane_fit_moments*, compute the *Plane* normal, residual (normal*covariance*normal / numpoints), and quality (gaussianness of plane fit) from the sums, compute the mean and variance of heights (z) of the summed points, and do I/O. Another function updates plane normal and residual if points have been added since the last such calculation.

- **Region** (map_grid – region.h) is a 2D transform (2D position and an angle), with virtual functions to set/get members, get a bounding box, test whether a point is in the region, do I/O, and transform by an angle and offset. Many of these are set to null to force the inheritor to define them. The class seems redundant with *Grid_Map_Origin*. **Circular_Region**, **Square_Region**, and **Quadrilateral_region** are children of *Region* with descriptive parameters (radius, halfwidth, or corners) and overloaded functions.

- **Edge** (Analysis_edge – Science_analysis.h) is a linked list node with vector of *Point_2D*. Probably redundant with crater detection code.

3.6 2D Grids

These are not exactly computer vision structures, but to the extent that vision is used to map the environment, you may have use for them.

- **Grid_map_origin** (map_grid – grid_map_origin.h, grid_map.cc) is 2D-transform (2d position and 1D orientation) with setters, getters, constructors, and functions to convert points and orientations between local and global coordinates. It should perhaps be replaced with a 2D-transform class.

- Indexers (map_grid – grid_map.h, grid_map.defs.h), specifically **Standard_grid_map_indexer** and **Wrap_around_grid_map_indexer** have functions that take an x or y coordinate and a height or width, and return the coordinate as-is or wrapped around the height or width. They also have a method to scroll a *Grid_Map* to put indices (r,c) in bounds, though the standard indexer just returns false.

- **Grid_Map** (map_grid – grid_map.h, grid_map.defs.h) is 2D grid, represented as an *Array_2D* of templated data type representing map cells. The class records the map dimensions, the width of each (square) map cell, and coordinates of a map corner, all in 2D world coordinates. Functions convert between a 2D point or rectangle and the array indices, corner coordinates, or center coordinates of the enclosing cell(s). These functions can use map or world coordinates, converting via a member *Grid_map_origin*. Constructors all wrap map_setup(). Additional functions get/set member variables; get/set a cell at array indices or 2D coords; get map bounds as coordinates; clear all cells using cells' clear() or by filling them with a parameter cell; get *Global_iterators* to step through map in x or y; shift the map to include a point; and do stream and file I/O. Virtual merge_maps() is declared but not defined. Requires an Indexer template type to assign to its iterators.

  One choice for templated data type is **Grid_Map_Cell** (map_grid – grid_map.h), which is a base class that just defines an empty function, clear(). It is never used, but presumably provides the minimum interface for a cell in *Grid_Map*. Perhaps *Grid_Map* should use *Grid_Map_Cell* instead of a templated type.

  Const_Grid_Map_Iterators and **Global_Iterators** (map_grid – grid_point_iterator.h, grid_map.defs.h) march a "current position" across a *Grid_Map*. **Coordinate_Iterator** has a 2D position and 2D step size, constructors to set them, and overloaded add/increment operators. It uses only map coordinates. **Global_Iterator** is a *Coordinate_Iterator* whose position and step (but not constructor) use global coordinates. **Grid_Map_Iterator_Base** is a

Coordinate_Iterator whose constructor (but not position and step) uses global coordinates. It also has a function to determine the iterator bounds – the smallest and largest number of increments that will place the iterator's position on a parameter, template typed, map. **Grid_Map_Iterator** and **Const_Grid_Map_Iterator** are *Grid_Map_Iterator_Base*s with a member *Grid_Map* pointer and functions to determine iterator bounds for the *Grid_Map*, test whether it is currently in bounds, and get the "current" *Grid_Map* cell.

- **Region_Iterator** (map_grid – region_iterator.h, region_iterator.defs.h) has a *Grid_Map* and a *Region*. It seems to be for iterating through the grid cells under a *Region*. It records the bounding box of grid cells containing the region and the array indices of, coordinates of, and pointer to, a "current" *Grid_Map* cell. The constructor converts the region from global to map coordinates, determines the grid cell bounding box, and sets the "current" cell to the top left cell in the bounding box. The increment operator moves the "current" cell along the bounding box in reading order until it finds a cell in the region or until it runs out of bounding box. There are also functions to get the current cell, the bounding box, an iterator at the top-left of the bounding box, or an iterator at the bottom left (probably an error) of the bounding box. There are operators to test whether two iterators have the same current cell.

- **Plane_Fit_Map** (analysis_terrain_morphin – plane_fit_map.h) is a *Grid_Map* whose cells are *Plane_Fit_Moments*. It has members to describe bounding box and clipping and a function (called by constructors) to fill them from a *Morphin_Analysis_Params*. It has virtual stream and file I/O functions and functions to write the map as a green-scale elevation map. It has a pass-through to clear the map and functions to add a point or a point cloud to the map. This adds the point(s) to the appropriate *plane_fit_moments* element(s) but does not recalculate the plane(s) there. It does update the *Plane_Fit_Map*'s bounding box. This is potentially more general than morphin, so perhaps the function to fill the parameters from morphin_analysis_params belongs with the latter.

- **Plane_Fit_Overlapping_Map** (analysis_terrain_morphin – plane_fit_map.h) is a *Plane_Fit_Map* that also has a *Region*, which is set in the constructor. It has overloaded functions to add a point or point cloud to the map. These functions center the *Region* at each parameter point, and add the point to all cells under the region. When adding a single point, you can specify an in-plane rotation of the region. The class has overloaded I/O functions. It has pass-throughs to find region bounds and determine whether a point is in the region.

3.7 Other Objects

Here are other classes that don't fit in above.

- **Targets** (Analysis_edge-R1-00a – Science_analysis.h) is a class with: arrays of row, col, size, and rank; setters; function to double array allocation; function to draw each (row,col) on an image. Variable Npts tells amount of arrays used, but is not updated by methods. **PointArray** (Analysis_edge-R1-00a – Specpoint.h) has the same data as *Target* but statically allocated to 100 elements each. *Point_Array* has no functions.

- **Bounds** (map_grid – grid_map_point.h) is an array of four point_2d.

- **Mesh** (arc_vision – mesh.h) includes an array of 3D points, an array of 2D texture coordinates, and an array of triangles, dimensions of each array, and a *Frame_l*. Can construct from a point cloud, taking every nth point (param n), though it is not clear whether this means every nth row as well as every nth column. Contains functions to write the mesh as an inventor file. **Mesh_io_vrml** (arc_vision – mesh_io_vrml.h) includes code to write a parameter mesh as a vrml file, adding member variables to specify material properties.

3.8 Non-vision objects that you might use anyway

The following data structures have precious little to do with computer vision. However, you may encounter them, so I've documented them.

- **Parse_Block** (String_io – parse_block.h) is an stl::map of "label = value" pairs, such as you might find in a parameter file. Major functions convert to/from a string or stream with format "{ label = value … }", add a label/value pair, check whether a label appears in the map, and get/set a value for a parameter label (getter has optional default value in case label is not present). Other functions get pointer to map, copy all labels into a vector, convert a label to a *String_Rep*, and clear the map. Construct empty or copy another *parse_block*. **Cmd_string** (String_io – cmd_string.h) is a *Parse_Block* whose first string/stream entry is a command name, stored internally as a value with label "_command". This class has functions to get/set the command value. **Cmd_Array_Entry** (string_io – cmd_array_entry.h) is an stl::vector with functions to read/write to string/stream with format "[blah   blah   blah …]" and to copy to a CLARAty *Vector*. Construct empty, or pass a number of elements, an stl or CLARAty vector, a string, or a data pointer.

- **String_Rep** (share – string_rep.h) is an stl::string that converts to a numeric type when you typecast it, so you don't have to manually atoi(), etc.

- String_io/ace_time_string.h as functions to read/write *Ace_Time_Value* and to convert it to/from string.

- **Semaphore** (share – semaphore_t.h) has an int protected by a mutex. I don't really get it.

- **Ref_count** (share – refcount_t.h) is an integer (default value = 1) that you can use as a counter. It has a getter and functions to increment/decrement. Mainly it is used to track how many variables point to the same block of memory, so that a destructor can tell not to free memory that is still in use.

- **Factory** (share – factory.h) is a class for helping other classes instantiate descendents when you don't know ahead of time which descendent will need to be instantiated. A *Factory* has an STL map of constructor pointers versus "keys", a function to add such a pair, and a function to pass in a key and return the result of the constructor. *Factory* works with class **single_instance**, which takes two functions as template args, calls the first function when a *single_instance* is constructed while no others are, and calls the second function when the last *single_instance* is destructed. To incorporate a factory into a base class, the class must declare a member Factory *, have its header file call a macro to make a static *single_instance* of a factory for that class, and have its cc file call a macro to instantiate the factory's static space. Each descendent that will use the factory must also call two macros to create/instantiate a class that adds the descendent's constructor to the factory and to instantiate that class' static space. It appears that *camera_model* is the only base class that wants a factory.

- **GL_Object** (util_open_gl – gl_object.h) is a base class for drawing. It has a *Point<float>* (world coordinates of object origin) with setters, getters, and functions to add another *Point<float>* to it. Has a do/not draw flag with setters and getter. Has a virtual (must be redefined) function to draw the object, a function to translate to the member point and then draw the object, and a function to draw text.

- **GL_Display_List_Object** (util_open_gl – gl_object.h) has an Opengl display list ID and functions to start, end, and execute the display list.

- **GL_Origin** (util_open_gl – gl_object.h) is a *GL_Object* whose draw function draw coordinate axes of (member variable) length starting at the object origin.

- **GL_Hash** (util_open_gl – gl_object.h) is a *GL_Object* whose draw function draws a 3-axis, 3D cross of (member variable) length and color starting at the object origin.

- **GL_Ground_Plane** (util_open_gl – gl_object.h) is a *GL_Object* whose draw function draws a grid of lines in the X-Y plane. Grid has spacing, number of cells, and line color according to member variables set in the constructor. Every 4th line has inverted color.

- **GLUT_Window** (util_open_gl – glut_window.h) is a wrapper class around GLUT to draw 3D scenes. It provides a static array of GLUT_Windows, which each store flags for mouse motion and mouse buttons and parameters for a window and its associated frustum. The class has a list of registered GL_Objects with functions to register, unregister, and draw all. It has various functions to modify the window and to override callbacks such as timer, key handling, and mouse motion. Has functions to draw text and to manipulate windows, such as hiding/showing, refreshing, and changing the title bar.

- **MemoryManager** (JPLPic – nav_memory.h) replaces standard memory allocation. Lots of comments in the file describe usage. The important part is that a class that uses a *MemoryManager* generally has a pointer to one as a member, and assigns it in the constructor. It can then use the NEW, DELETE, and DELETEV macros, which take the pointer, instead of new and delete, to manage memory in a safe manner. Seems like those should be member functions instead of macros, but they are not. *MemoryManager* also provides functions to look at memory usage. The file redefines new and delete, unless you specify NAV_MEMORY_BYPASS, which *JPLPic* does.

4. General CLARAty Vision Functions

This section describes objects and functions that feel more like operators than data structures. I have somewhat arbitrarily grouped them as image operators, image I/O, feature detection/tracking, visual odometry, stereo, hazard mapping, miscellany, and non-vision functions that I felt compelled to document. For a good overview, just skim through and look at the boldface headings.

4.1 Image Operators

Image Operators are those operations that, for the most part, apply in the same way to each pixel in an input image. They are supposed to inherit from class Image_Op and have a common interface. Currently nothing inherits from Image_Op, but most share Image_Op's interface.

- **Image_Op** (Image_Ops – image_op.h) is a base class intended to be inherited by the other image operations. It just declares a constructor and the function filter(). The latter is intended to take a source image and a destination image, resize the latter, and fill it by processing (but not modifying) the former. It will probably either become the base class for all image operators or it will go away.

- **Convolve_Op** (Image_Ops – convolve_op.h) has filter() that convolves the input image with a kernel and stores the result in the output image. Mathematically, it is actually a correlation, but image-processing people call it a convolution. There are some optimizations that kick in for 1-dimensional and/or anti-symmetric kernels. Specify the kernel and whether it is anti-symmetric in the constructor or with a setter. Pixels within ½ kernel of the edge of the input image cannot properly serve as centers for the kernel because part of the kernel extends beyond the image. Use the constructor or setters to specify whether the output image should have the same size as the input image, or be reduced by ½ kernel on each side (to avoid processing the borders), or be extended by ½ kernel on each side. For same sized or enlarged outputs, similarly specify whether to leave border pixels unprocessed, or whether to process them by inventing input image data by reflecting the edge of the input image, wrapping around the far side of the image, or setting off-image pixels to some constant value. Not all of these options are currently supported.

- **Gradient_Op** (Image_Ops – gradient_op.h.) Filter() convolves an image with a horizontal or vertical step or difference-(derivative?)-of-Gaussians mask, to generate a gradient image. Specify the mask type, direction, and size in the constructor or setters. Filter() actually calculates a convolution mask and then calls (inherited) convolve_op's filter() to do the convolution. It might make sense to not regenerate the convolution mask every time you filter(), though the time savings would probably be insignificant compared to the convolution time.

- **Rescale_Op** (Image_Ops – rescale_op.h.) Filter() rescales the input image intensity to the range between member variables "_min" and "_max." Set min and max in constructor or using setters. Filter() is not optimized.

- **Transform_Op** (Image_Ops – transform_op.h.) Filter() applies a 3x3 homogeneous transform to input pixel coordinates to re-map pixels to the output image. The output image has the same dimensions as the input image. Output pixels with no corresponding input pixels are set to 0. Set the transform in the constructor or with a setter.

- **Rectify_Op** (Rectify_Op – rectify_op.h.) Filter() applies a lookup table to input pixel coordinates to re-map pixels to the output image. There are actually two lookup tables, which are *Image<float>* that store the replacement x and y coordinates of each pixel. Fill the lookup tables in the constructor or with setters, by passing two lookup tables, a camera model (to create un-distortion tables), two camera models (to reproject from one distorted viewpoint to the other), a 2x2 or 3x3 transform, a 3x3 transform plus a camera model (to do un-distortion plus transformation), or another rectify_op and a subsampling factor. Global function **rectify_image()** does the same as member filter() but can also scale pixel intensity using a parameter function. **Subsample_rectify()** "decimates" the input image by an integer factor, and then applies the transform.

- **st_warp_*()** (camera_model_jpl – st_warp.h) are functions to generate, apply, and do file-I/O for warp maps, which are pixel coordinate look-up tables to rectify images taken with a CAHVOR(E) camera to equivalent CAHV model images. The functions rely on stereo.h, which does not appear in the module. The warp maps are stored as char *, presumably because C has no inheritance. I recommend moving the relevant structures from stereo to *camera_model_jpl* and making a warp map object.

- **Non_Maxima_Sup_Op** (Image_Ops – non_maxima_sup_op.h.) Filter() sets output image pixels to 1 if the corresponding input pixel has larger value than its 4-neighbors or 0 if it does not. A 1-pixel border has undefined values. There is a lot of commented-out code, implying that this function is not finished.

- **Corner_Detect_Op_Harris** (Corner_Detect_Op – corner_detect_op_harris.h) is the filtering step of a Harris corner detector. Filter() sets each output pixel (x,y) to the lower eigenvalue of the matrix [Σ(IxIx) Σ(IxIy); Σ(IxIy) Σ(IyIy)], where sums are taken over pixels in a window around coordinates (x,y) and Ix and Iy are input-image gradients at those pixels. Gradients are taken using a derivative of Gaussian mask with a 1-pixel sigma. Specify the window side-length and mask width in the constructor or with a setter. For additional speed, modify the code to not zero the output pixels before filling them and to use marching sums in the final loop.

- **Resample_Op** (Image_Ops – resample_op.h) has resample() that takes parameter "ratio" (floating point) and magnifies the input image by that factor. It takes an optional parameter for interpolation, which can specify any interpolation type understood by *Image*. There is also halfsample(), which is a shortcut to shrink an image by half, averaging 4-pixel blocks. Unlike arc_vision sub-samplers below, resample() can magnify or shrink an image and is not limited to integer resolutions. However, resample() has a non-standard name (should be filter()), takes too many parameters, and incorrectly assigns output pixels based on input_pixel*ratio rather than input_pixel/ratio. Also, resample_op is pretty clearly unfinished.

- **blend_subsample()** (arc_vision – blend_subsample.h) is standalone function that sub-samples an image, replacing each *n* by *n* block (integer *n*) with the average of the pixels in the block. It stores the results in a second image, leaving the first image unmodified. Assumes image type can cast to integer. If image size is not a multiple of *n*, output excludes incomplete blocks at the bottom and right edges, even though the function spends time making sure pixels are on the input image.

- **pick_subsample()** (arc_vision – pick_subsample.h) is standalone function that sub-samples an image, replacing each *n* by *n* block (integer *n*) with the pixel in the top left corner of the block. It stores the results in a second image, leaving the first image unmodified. If image size is not a multiple of *n*, output includes representatives of the incomplete blocks at the bottom and right edges.

- **Edge_Detect_Op_Canny** (edge_detect_op – edge_detect_op_canny.h) is a standard, 3-step Canny. It's Filter()generates an edge magnitude image (rms of 1D derivative of Gaussians), suppresses local non-maxima (complicated formula over 9-neighborhood), and threads adjacent edge pixels. Filter() takes an extra parameter (a pointer to fill with a list of edges) and returns an image of edge orientations. If the internal edge magnitude image, edge orientation image, and edge list were members with getters, then the class could be an Image_Op with standard filter() interface. If switches allowed the user to disable edge list and/or orientation image generation, then the user could avoid a speed penalty for any unused functionality. The threader, follow_iter(), follows adjacent pixels from a starting pixel to a pixel with no unthreaded 8-neighbors with edge magnitude above a "low" threshold. It stores the pixel positions in a list and copies their edge-magnitude-image values to the destination image. The threader ignores branches and pixels on the far side of the initial pixel, but it could track all branches if its inner loop did not break after finding one suitable adjacent pixel. Alternately, the existing functionality could probably be faster if the current pixel were not stored in a vector. Function follow() is a recursive, probably aborted threader that does not create edge lists. Filter() calls follow_iter() to create an edge vector for each pixel that does not already appear on another vector and whose edge magnitude is above a "high" threshold. It returns a list of all such vectors that have at least 3 pixels. Perhaps minimum edge length should be a constructor parameter or have a setter. The destination image and orientation are zeros for suppressed non-maxima and non-zero for maxima, even if associated edge vectors are eliminated. The constructor can set gradient filter size and the "high" and "low" threading thresholds, or it can accept defaults. Function _hypotenuse() should perhaps be replace the similar cl_rms_op(). The canny edge detector in the crater detection code should be merged with this class.

- **Fast, custom image ops on JPLPics** (jplpic – nav_fast_filter.h). Macros BEGIN_SCAN_ONE_IMAGE and END_SCAN_ONE_IMAGE wrap a kernel to operate on each pixel of a *JPLPic*. There are similar macros for TWO, THREE, and FOUR images. APPLY_FILTER_TO_ONE_IMAGE applies parameter filter-code to each pixel neighborhood in a *JPLPic*. Presumably these macros run much faster than (more elegant) inline functions that take a function pointer with the kernel code, because presumably you can't inline a function that you pass as a pointer. END_FILTER_ONE_IMAGE and COMPARE are #defined to nothing.

- **Fast, sliding-sum operators on JPLPics** (jplpic – jplpic.h, pic_filter_mem.cc) declare statically allocated, very large buffers for sliding sum operators, along with the operators that use them. Several operators all share the same space, instead of having separate buffers for each operator. Operators do blob filtering, smoothing (boxcar, questionable use of "scale" variable) and DoG (two boxcars) with optional decimation by ½ (decimate vertical, convolve, decimate horizontal), and gradient finding (x, y, both, or both plus smoothing). Most of those can produce byte or short images. Defines PREPARE_STATIC_BUFFER to wrap new and some functions to free all of the static memory.

- **Other, pre-defined JPLPic operators** (jplpic – jplpic.h and pic_filter.cc). #Defines INT_MAX and INT_MIN, which are surely redundant. Ifndef OMIT_IMAGE_OPERATORS: defines MiniMax (4 variants – seem to clamp an image at a minimum value of image and either scalar or other image pixel or neighborhood defined by mask image); has functions to shift and scale a *JPLPic* by two scalars or images; find the absolute difference between two images; convolve with another *JPLPic*; superimpose one image onto another even of different data type; and average the pixels in a mask-defined footprint. Has functions to rescale pixels (with optional param min/max) to an 8-bit image; resample (average) by arbitrary factor; subsample (decimate) by factor of 2, 4, or 8; extract a band and colormap from a multi-band image; create a left-to-right mirror image; compute a threshold at a histogram fraction; threshold into a new, 0-or-255 image. Many of these operators rely heavily on macros, making them illegible but perhaps fast. PyramidLevel is a wrapper for PYRLEVEL_GOOD_LOWBITS, and I haven't figured out what that does. Don't know what FindSun does, but it is in the "OMIT" group. There are declarations but no definitions of for (probably vestigial) functions to convolve with DOG filter, "decimate" by a factor of 2, and rotate an image. Pic_filter.cc includes stereo.h to get macros NO_RANGE and NO_S2_DISP, which are used in Resample() to avoid blending invalid pixels in 16-bit disparity maps and 3-vector float and double images. That is the only reason the JPLPic module has stereo.h.

NO_S2_DISP is redundant with JPLPic.h's IS_UNDEF_DISPARITY() macro, while NO_RANGE probably *ought* to be redundant with IS_UNDEF_3D_POINT(), though currently the latter is probably in error. Perhaps you could define the macros (or make them consts) here or in JPLPic.h instead of in stereo.h, and then eliminate stereo.h from the module. A local copy of stereo.h is no more robust than local macros. Further, perhaps the stereo code should get its definitions of NO_S2_DISP and NO_RANGE from JPLPic.h, not vice versa, as they are a feature of the image, not the stereo algorithm. IS_UNDEF_3D_POINT may be working with the pad value of JPLPic::WriteHeightField(), not the undefined value. Perhaps those two should be made to match.

- **Filter_Slog** (arc_vision – filter_slog.h) implements the Ames SLOG algorithm. Apply() convolves an input image with a Gaussian, convolves the result with a cross filter (4 at center, -1 at each 4-neighbor), then thresholds the result, setting negative values to 0, non-negative values to 255. It stores the results in a second image, leaving the original unchanged. It can handle a sub-image as the input image. A 1-pixel border (where the cross filter cannot reach) is all 0, and a larger border (where the gaussian cannot reach) has undefined output. There is a setter function for the width of the Gaussian (specify root2*sigma). Internally, the class has a Gaussian mask maker and a convolver specialized for a symmetric, separable mask. Both internal functions are or should be redundant with the CLARAty Convolve_Op or its descendents. The convolver appears to handle the last row of data incorrectly as it cleverly convolves in place. There are several opportunities for optimization, including copying in a sub-image, not zeroing the entire output image, and not convolving the margins of the image.

- **Correlator** (arc_slog_tracker – correlate.h) is a fast correlator for 1-bit-per-pixel *ATImages*. Its single regular function, Correlate32(), XORs a (parameter) 32x32 kernel across a (parameter) sub-window of a (parameter) input image, storing the number of common bits under the mask (0..1024) in an output image. The correlator can skip (a parameter number of) rows and/or columns between mask positions. The kernel is passed as 32 *ATImages*, each shifted 1 bit from the previous one. The class uses a static lookup table to count the number of common pixels (zero bits) in two XORd shorts. It has a function to generate the table and a static counter to make sure that the table is not re-constructed or destructed while another Correlator is using it.

- **Analyzer** (arc_slog_tracker – analyze.h) has functions to find the maximum value in a (parameter) sub-window of an (input) 16 or 32 bit-per-pixel *ATImage* or the minimum in a sub-window of a 32 bit-per-pixel *ATImage*. Not sure why the functions are not templated for pixel type.

- **Transformer** (arc_slog_tracker – transform.h) has many operators for *ATImages*. There are functions to filter a sub-window of an image into another image, using a (parameter size) moving average filter or an SDOG (sign of difference of 2 moving average filters of parameter size). There are functions to create a new *ATImage* by: cropping an *ATImage*; replacing an image with 7/8 of its intensity + 1/8 intensity from another image; packing a 16 bit-per-pixel image containing only 1s and 0s into a 1 bit-per-pixel image; shifting a packed image by 1 bit; rotating the image; subsampling a subwindow at an integer period; and finding sign or magnitude of difference of two images. Some functions implicitly require 16 bit-per-pixel images.

- **Dewarper** (arc_slog_tracker – dewarper.h) has a principal function that unrolls an x-y *ATImage* into a rho-theta *ATImage*. A member lookup table records the input pixel address for each output pixel. A Configure function creates the lookup table. Construct empty and call Configure() yourself (safe) or pass parameters to a constructor that calls Cofigure() (convenient.) The class records dimensions of input and output images, and the center and radius of the input image disk to be unrolled.

4.2 Image I/O, Display, and Markup

These are non-vision image operators. Image I/O consists of subclasses of classes *Image_IO* and *RBG_Image_IO*, all sharing a common interface. Image display generally uses QT, which is apparently a way to quickly generate X windows. I've never gotten it to work in CLARAty, and the documentation here is not enough to use it.

- **Image_IO** (Image_IO – image_io.h) and **RGB_Image_IO** (Image_IO – rgb_image_io.h) are base classes for file I/O with *Image* and *RGB_Image* objects. Most other classes have their own I/O routines, but with the large number of image file formats, image file I/O is big enough to warrant its own class. The class has a member filename string and declares one pair of load() and save() functions for each standard pixel data type. The functions are virtual and defined to null, so that a child class is forced to overwrite them. Explicitly declaring functions for each pixel type is ugly, but perhaps it allows us to not templatize the *Image_IO* and *RGB_Image_IO* classes and yet accommodate compilers that can't nest templates. Each load() and save() function actually has two declarations, one taking a filename and the other not. Why not just make the parameter optional and have just one declaration?

- **PNM_Helper** (Image_IO_PNM – image_io_pnm.h) is a class with functions to read/write PNM headers and comments to/from streams. The functions are static, so you don't need to declare a PNM_Helper, just use it as a scope operator. The same file defines enum **PNM_FILE_TYPE** (the 6 pnm types) and enum **PNM_FILE_INFO** (the information in a pnm header.) You can use these in functions that read/write pnm files.

- **Image_IO_Pgm** (Image_IO_PNM – image_io_pgm.h) is a child of *Image_IO* that redefines load() and save() to load P2 (ASCII) or P5 (binary) PGM files and write P5 (binary) PGM files. The class includes a string for in-file comments, which are read/written along with a file. **Image_IO_Pgm_Simple** (Image_IO_Pgm_Simple – image_io_pgm_simple.h) is an apparently obsolete child of *Image_IO* that reads and writes only P5 PGM files.

- **RGB_Image_IO_PPM** (Image_IO_PNM – rgb_image_io_ppm.h) is a child of *RGB_Image_IO* that redefines load() and save() to load P3 (ASCII) or P6 (binary) PGM files and write P6 (binary) PGM files. The class includes a string for in-file comments, which are read/written along with a file.

- **Image_IO_Pnm_State** (camera_image_io – image_io_pnm_state) is a class for reading/writing PNM files. It is presumably redundant with the image_io_pnm module, and should probably get merged. It has 3 functions to load, save, and load_stereo. The class also has an internal pnm_header class, which can read and store a header and has a function to read an image body. Image_state.h has 3 inline global methods that each create an *image_io_pnm_state* and call one of its three functions, so you needn't explicitly instantiate an object. The loader and stereo loader do a lot of confusing *Frame* handling.

- **Image_IO_Tiff** (Image_IO_Tiff – image_io_tiff.h) is a child of *Image_IO* that redefines load() and save() to read/write tiff files using a bunch of commands from <tiffio.h>. **Image_Tiff_IO** (Image_Tiff_IO – image_tiff_io.h) is an obsolete version of *Image_IO_Tiff* that does not use const for function parameters or use a Tiff error handler during file writing.

- **JPLPic File I/O functions** (jplpic – jplpic.h, pic_io.cc) include functions to: read/write a PIC file with optional endian reversing (several variants); write an image as floats; guess a file's pixel type from its name; read/write arbitrary file format (using functions from libimage and parse_image.h ifdef NEW_READ, else only works with PIC files); write with an optional overlay to a rawbits-PPM file; write as ASCII art in 16 "grey" levels and user-specified dims; and write to screen/file as atoi'd text. Has functions to read an image from a file with a C++ string-format version (with backslashed characters and octals for unprintable characters), and to convert between image and C++ string in memory. If OMIT_DRAW_OPERATORS is undefined but EXTRA_JPLPIC_WRITES is defined, has function WriteGif. Helper functions (jplpic – good_fopen.h) determine whether a file or directory already exists, locate a file on a set of paths, and open a file for write even if the file already exists, and . These are good things to make generally available in CLARAty, but I don't know where.

  JPLPic has a member variable that can combine three #defined bit flags (defined in jplpic.h) to specify conversions to apply during writing. The conversions are: scale intensity to 8 bits-per-pixel, convert floats to doubles, and byte swap for floats/doubles. There is a getter for the variable and setters that clear all flags or set/clear one. The constructor initializes the variable to parameter flags.

  Pic_io.cc () defines NO_RANGE to understand missing pixels when writing a height map and invalid pixels when writing a 3-vector image text 3-vectors. It may also get confused with the 10000 appearing elsewhere in the code. It would be better to use the same NO_RANGE that pic_filter.cc uses, which probably is based on IS_UNDEF_DISPARITY() in JPLPic.h or NO_RANGE declared independently in stereo.h. Neither option is particularly likely to keep up with changes in the stereo code that produces these pixels, but at least the different parts of JPLPic will be internally consistent.

  The I/O routines use a few supporting functions/structures/macros (jplpic – image_parse.h, filename.c, libimage.a). Prototypes for grey_image_read/write and color_image_read/write, which mainly are used by pic_io.cc's Read/Write GenericImage(), appear in image_parse.h, with definitions presumably in libimage.a. Those functions use **ImageMemoryT** (which basically duplicates *AnythingT*) and *ImageScaleT*, both from the same header file. The files also declare (but do not define or do not use) a lot of structures/functions for operations such as handling filename suffices, converting between suffices and file formats, and reading/writing range images. JPLPic includes image_parse.h, but probably needn't, because it probably only uses IS_TRUNCATE, and it probably needn't. Filename.c relies on mwm.h only for safe_strncpy(), which is a little silly.

PicHypothesisT (jplpic – image_parse.h, filename.c) is an image header, apparently used for file I/O. Along with find_pic_params(), it is used in pic_io.cc's FilePixelType() to guess the *JPLPic* enum PixelType value most appropriate for a file. The function is undefined, but may appear in libimage.a.

FormatTagT (jplpic – image_parse.h, filename.c) holds information about an image file format: a string with the standard filename suffix; the corresponding value of enum **ImageFormatT**, which has one value per file format; and pointers to functions for reading, writing, and testing whether a file is that format. Array *known_image_formats* has one *FormatTagT* for each value of enum *ImageFormatT*. It is filled in with lots of format names and pointers to declared-but-undefined functions. A function (used by pic_io.cc's WriteGenericImage()) guesses the proper format for a filename, helped by a function that sorts through *known_image_formats*. There is an unused function to print a list of formats from *known_image_formats*. Seems like all of this belongs as one object.

ImageT (jplpic – image_parse.h) is another image format. It has rows, cols, four data pointers (red, green, blue, grey), and several fields that appear to be for file I/O. It is only used by free_imageT() (undefined) and four file I/O functions defined in libimage.a. The I/O files are only called from pic_io.cc's Read/WriteGenericImage(), and only ifdef NEW_READ.

- Here is an idea for loading images without knowing their file format a-priori. It is redundant with (jplpic – filename.c)'s unused find_format_from_file().
    - Let *Image_IO* declare, "virtual bool check_magic_number (char *) =0," and have sub-classes of *Image_IO* define check_magic_number() to tell whether the parameter string contains the "magic number" for the class' file format.
    - Make **Image_IO_generic_read**, a sub-class of *Image_IO*. Give the class a list of *Image_IO* pointers and a function to add pointers to the list. Redefine load() to open a file, alternate calling check_magic_number() and rewind() for each *Image_IO* pointer on its list until one returns true, then close the file and call that Image_IO's load(). To use *Image_IO_generic_read*, programs must instantiate and register pointers to subclasses of Image_IO for each file format he wants to be able to read. This strategy does not address pipes, which cannot rewind, and it does not address saving to multiple formats.
    - To optimize the process, have Image_IO sub-classes separate _load() into check_magic_number() and _load(), have their load() call both functions,

and have Image_IO_generic_read::load() call the proper _load() rather than closing the file and calling a sub-class' load(). Subclasses that handle multiple formats, such as *Image_IO_Pnm*, could record the actual format in a member variable as part of check_magic_number().

- **Draw_Op** (draw_ops – draw_op.h) has functions to draw a box or a cross of parameter size on a parameter *RGB_Image*, centered at parameter coordinates. You can pass a parameter *RGB_Color* or default to a member variable color. The module appears unfinished. It would be a good module to host the ellipse drawing routines in the crater detection code.

- **Cross_hair()** (Analysis_edge – Science_analysis.h) takes an image and a list of ellipses, and draws cross hairs and dotted ellipses on the image at each listed ellipse. The crosshair drawing probably belongs with Draw_Op. The ellipse drawing probably is redundant with the crater detection code.

- **DrawFOV()**(analysis_edge – specipoint.cc) draws a black circle of given center and radius, plus a point at the center and an optional diameter in a given direction.

- **Transformer** (arc_slog_tracker – transform.h) includes functions to draw a box or cross on an *ATImage*.

- **JPLPic Drawing routines** (jplpic – jplpic.h and pic_draw.cc). Macro SET_PIXEL_AND_INC fills a pixel (and increments its pointer) from a color map index or by shifting the existing color. Several functions use SET_PIXEL_AND_INC to draw on a *JPLPic*: fill border with a fill color; set one pixel's color; draw an empty/filled box; draw an arc/circle (3 variants) using many macros and optional fill. If OMIT_IMAGE_OPERATORS is not defined, there are also functions to truncate a line segment to the part that appears in the image, draw a line segment, draw a text string (5x8 font) or integer (5x7 font), and get the pixel dims of a text string. #defines ABS(). Most drawing functions have variants that allow wrapping around the image border. There is a function to test whether a colormap entry just repeats one byte, and if so to return that byte. That probably doesn't belong here, but it is in this file. Many of these functions are overloaded instead of using default parameter values. It is not clear why only some of the drawing functions depend on OMIT_IMAGE_OPERATORS. Functions to draw an empty/filled polygon are declared but undefined.

- **Image_Displayer** (image_displayer – image_displayer.h) uses QT and threads to pop up and write to displays (i.e. windows.) The constructor opens one display. A function can open up to 10 total. Other functions: copy an *Image*, *RGB_Image*, or char * plus two dims into a display; make a display non-resizable (cannot undo); and query coordinates of last mouse press in a display. Displays are stored as **Image_Display** (image_displayer – image_display.h), which has the code to set display images and to handle the display's repaint and mouse events. The relationship between *Image_Displayer* and *QtImageWidget* is unclear, but probably the former shows more images and the latter does more with one image.

- **QtImageWidget** (qt_image – qt_imageviewer.h) is a *Qwidget* whose paint-event draws a member *Qimage*, at a member scale (default=1), along with a member list of *ImageAnnotations*. It has functions to disable scaling, determine current size from image dims and scale, tell the display to redraw without erasing, and set the member variables then redraw. You can set the *Qimage* member to an *Image* or a *Qimage*. There are handlers for mouse click and move, and for repaint. **ImageAnnotation** (qt_image – imageannotation.h) is a text ornament for an image. It stores 2D coordinates, a string, and a bool to say whether to show a cross. It has getters. **Qt_Image_Subscriber** (qt_image – qt_imageviewer_subscriber.h) is another QT class. It has a *QtImageWidget* and appears to periodically query and redraw its *Image*.

- **QtImageViewer** (qt_image – qt_imageviewer.h) is a *QtImageViewerUI* (undefined class) with a *QtImageWidget*, an *Image*, and a gamma correction. The constructor seems to open a display window and attach default buttons to member functions. There are pass-throughs to set the *QtImageWidget's* image and annotations. There are functions to zoom the image, write labels giving 2D coordinates or image dims+scale, draw the image to the display, respond to mouse click/move, and modify gamma correction. **Image_Updater** (qt_image – qt_image_update_tester.h) has a constructor that sets a parameter *QtImageViewer* to refresh using a parameter *Image* at a parameter frequency.

4.3 Feature Detection and Tracking

We implement a Harris feature detector, a Lucas/Kanade/Shi/Tomasi tracker, a correlation-based tracker, and a SDOG tracker. We use a feature class with various specializations.

- **Feature_Window** (feature_tracker – feature_window.h) is a 2D pixel window extracted from an image pyramid, suitable for Shi-Tomasi tracking. It records the window's initial 2D position in the full resolution image, the number of pyramid resolutions, a copy of the window's pixels at each resolution, and the Shi-Tomasi gradient images and sum-of-gradient matrices for each resolution. The regular constructor allocates/fills the member variables from an image pyramid, initial 2D position, and window width and height. The constructor uses a 7-pixel mask for image derivatives and does not put feature windows within 3 pixels of the border. There is also a copy constructor.

  Descendents of *feature_window* define additional members (*e.g.*, rotation angle) to record accumulated translation and deformation of the feature. They define functions to calculate image gradients and sum-of-gradient matrices with respect to the new members, convert feature window coordinates to deformed/translated feature coordinates, and add sets of those members. **Feature_Window_Trans** (feature_tracker – feature_window_trans.h) is a *feature_window* with a 2-vector for translation. Three descendents of *feature_window_trans* are **Feature_Window_Scale** (feature_tracer – feature_window_scale.h) with a variable for scale, **Feature_Window_Zrot** (feature_tracker – feature_window_zrot.h) with a variable for in-plane rotation, and **Feature_Window_Affine** (feature_tracker – feature_window_affine.h) with a 2x2 matrix to handle affine deformation. These three use member *Images* as lookup-tables to expedite get_Jg(). *Feature_Window* defines a sum-of-gradient finder function that only *feature_window_trans* uses, as all other children overwrite it. Perhaps the function should move to *feature_window_trans*. The four functions defined by each descendent should perhaps be declared virtual in the base class so that feature detectors and trackers could operate on a *feature_window* pointer instead of individual child types.

- **Feature_Detector** (feature_tracker – feature_detector.h) has functions find() that detect and append good features from a parameter image onto a (possibly empty) parameter list of features (of one child class of *feature_window*). Perhaps one find() could take a list of *feature_window* instead. Find() works in five steps. First, it applies a *Corner_Detect_Op_Harris* to the input image to find the image of lower-eigenvalues. Second, it zeroes any eigenvalues below a threshold, which the user can set to a fixed value or a fraction of the highest lower-eigenvalue (default is 50%.) Third, it zeroes any eigenvalues that are smaller than any of their 8-neighbors, leaving only local maxima and plateaus. The current formulation can produce some artifacts because it modifies the lower-eigenvalue image in place and because it does not filter or zero a 1-pixel border around the image. Fourth, it zeroes any eigenvalues that are within 7 pixels of a non-zero eigenvalue or an existing feature (to prevent close-together features that a tracker could jump between.) Finally, it generates an image pyramid, constructs feature windows, and adds them to the list. The constructor takes feature window size, sizes for derivative and corner filters, and minimum distance between features, though it has defaults and setters for each. The minimum distance between features is not actually used, as the constructor hardcodes a value. A tex file in the module directory has a detailed algorithm description.

- **Feature_matcher_lk** (feature_tracker – feature_matcher_lk.h) is a Lucas/Kanade/Shi/Tomasi feature tracker whose function match() tracks a list of *feature_windows* into a new *Image*. Match() uses generic equations and virtual functions from descendents of *feature_window* to allow arbitrarily deforming features. There is one match() for each child class of *feature_window*, each calling internal _match(). Why not just have one match() take a list of *feature_window*? Setters control whether to end after number of iterations or fixed threshold (default 20 iterations.) Match() creates an *image_pyramid* for the new image, then for each resolution starting at the lowest, it adds the *feature_window*'s position and translation to get new position, scales to current resolution, then iteratively updates the position until reaching the stopping criterion mentioned above. The code is set up to reject features but, despite calculating the SSD each iteration, it has no test to actually reject any features. A tex file in the module directory has a detailed algorithm description.

- **Feature_matcher_bf** (feature_tracker – feature_matcher_bf.h) is a "brute force" correlator that tracks a list of *feature_window_trans* into a new *Image*. It updates the translation and SSD fields of each feature to reflect the minimum SSD of the correlation within a search window. Search window size defaults to 20x20, but can be set in constructor or with setters. The window is truncated at image bounds. Image access by operator(r,c) is probably slow. Average SSD is overestimated at image edges, where window is smaller but divisor is constant. Does not implement sliding sums. Does not do sub-pixel parabola fit. Does not do image pyramiding.

- **Tracker** (arc_slog_tracker – tracker.h) is an SDOG tracker, which is a fast correlator that preprocesses one feature template (kernel) and a correlation window in the novel image (area of interest – AOI) with a "sign of difference of Gaussians" operator that reduces each to one bit-per-pixel and packs them into 32 pixels per long integer. It correlates these by summing absolute difference of 32 pixels at a time using one or two XORs and table lookups. Two member variables of type *TrackerParams* store the size, position, and other details of the kernel and the AOI. They can be set in the constructor or by calling member Reconstruct(). The 32x32 kernel is stored as an unprocessed *ATImage* (_crop) and as a set of 32 SDOG'd, packed *ATImages* (_packed_kernel[]), each shifted 1 bit further to the right to allow quick correlation starting at each of the 32 pixels. The kernel is created from an *Image* and kernel-center coordinates using member functions sdog_create_template_32() to set the kernel or sdog_update_template_32() to average the new kernel (1/8 weight) with the previous one (7/8 weight). Member function sdog_track() extracts the AOI at parameter coordinates in a parameter *Image*, reduces it to an SDOG'd, packed *ATImage*, correlates the kernel across it, and returns either the resulting *Image* or the coordinates and height of the correlation peak. Internally, the tracker uses a large bank of member *ATImages*, converting to and from *Images* as necessary at the beginning and end of functions. *Tracker* is a child of *Visual_Tracker*, which may be in the yet-undefined visual_tracker module. It has several functions that pass through to the parent class.

  TrackerParams (arc_slog_tracker – trackerParams.h) holds parameters for an image sub-window to be used by a feature tracker: window dimensions, window coordinates (top-left corner and feature "center" within window), the widths of two masks for a difference of Gaussians (actually difference of boxcars) operator, and a sub-sampling rate to support reduced resolution tracking. Pass the parameters to the constructor or later to Init(). Has functions to re-set the feature top-left corner coordinates and to clamp them within a (parameter) distance of image border, presumably both to account for a new sub-sampling rate or a feature that has moved. Has a function to write all members to stdout.

checkRange() (arc_slog_tracker – trackerUtils.h) clamps the value pointed to by a (parameter) pointer at the range (parameter) min..max. Presumably this is vestigial or soon-to-be used code for the tracker.

TrackerResult (arc_slog_tracker – trackerResult.h) holds the output of a tracker, with member variables x, y, and confidence. Constructor initializes the members. Presumably this is vestigial or soon-to-be used code for the tracker.

- **Camera_tracker** (arc_slog_tracker – camera_tracker.h) is a child of *Tracker*, specialized to take images from a camera and leverage estimated camera motion. It seems to be still in development. Member variables record the coordinates of the feature being tracked (i.e. the correlation peak) and the height of the correlation peak. Function init_template() acquires a *Camera_Image* via member _camera (presumably defined and initialized elsewhere), calls inherited sdog_create_template_32() to create a template around a parameter *Point_2D,* and fills member correlation peak coordinates and height from with the *Point_2D* and 1024 (maximum height.) Function match() acquires a *Camera_Image*, correlates with the kernel using sdog_camera_track(), and records the correlation peak coordinates and height in the member variables. Sdog_camera_track() actually just records some details about the *Camera_Image*'s camera and calls inherited sdog_track(). Match() centers the correlation at the previous correlation peak coordinates (but see below) and reads correlation window size from (apparently inherited) *Camera_Tracker_Param* _param. If the correlation peak height is between the "update" and "replace" thresholds (also stored in _param), match() calls sdog_update_template_32() to modify the kernel. Various functions in *camera_tracker* provide a lot of debugging, writing images to file, and drawing images to GUI.

  Variable _targets[0] (presumably defined and initialized elsewhere) may contain 3D coordinates, presumably corresponding to the projected *Point_2D* being tracked. If so, then match() changes behavior slightly. First, it centers the correlation window at the projection of _target[0] into the new *Camera_Image* (using the *Camera_Image's* frame transform) rather than at the previous correlation peak. Second, if match() finds a new correlation peak height above the "update" threshold, it moves _target[0] to the nearest point on the ray from pinhole through the new correlation peak, implying that the correlation is more accurate than the motion estimate. Third, match() has an unexercised option, to record the distance from pinhole to _target[0] in member _original_distance (initialized by init_template()), and scale images to maintain the constant apparent distance to (and thus size of) the target.

  Camera_tracker_Telem and Camera_tracker_Param (arc_slog_tracker – camera_tracker_telem.h) are children of *Visual_tracker_telem* (undefined but probably in visual_tracker module). The former has a public string. The latter has four public variables, recording a search window size, a confidence threshold to modify the kernel, a confidence threshold to replace the kernel, and a flag saying whether to use a detailed GUI. Both classes define a bunch of overhead that seems to involve file I/O.

Tracker_Telem and Tracker_Param (arc_slog_tracker – tracker_telem.h) are children of *Telemetry* (undefined but probably in visual_tracker module). The former has x, y, and confidence, as if it were a *TrackerResult*. They redefine a few virtual member functions. Both classes appear to be unused, though they are presumably related to other classes with similar names, and the include file is (needlessly?) included by arc_slog_tracker – camera_tracker.h

- **Multi_Slog_Tracker** (arc_slog_tracker – multi_slog_tracker.h) is a child of *visual_tracker*. It has an array of *Camera_Trackers*, allocated using member reset(). Member init_template() takes a list of *Point_2D*, pairs each point with a *Tracker* (assumes same number of each), and calls each *Tracker's* init_template() to make a kernel from each point. Member match() calls each *Tracker's* match() to track one point, then does a bunch of image saving and drawing to GUIs.

4.4 Visual Odometry

There are three modules related to visual odometry. One is a generic wrapper class, one is a specialization using the JPL code, and one is a tester. These classes will soon be revised, so I'm not yet going to document them.

4.5 Hazard Mapping

Morphin is an algorithm to generate a traversability "goodness" map from stereo data. We have a wrapper class for the algorithm, a class for the goodness map, two classes to describe the cells of the goodness map, and one class to store the parameters to Morphin. Why not combine the morphin parameters and function classes? Why not combine the two goodness map cell classes?

- **Goodness_Cell** (analysis_terrain_morphin – goodness_map.h, goodness_map.cc) is a *Grid_Map_Cell* with fields for height, certainty, and "goodness". It redefines clear() to zero the extra fields. It has assignment and stream I/O operators. It also has a function to convert certainty and goodness into an RGB value, interpolating from green at goodness=1 to yellow at goodness=high threshold to red at goodness=low threshold, all with intensity proportional to certainty.

- **Goodness_Map_Template** (analysis_terrain_morphin – goodness_map.h, goodness_map.defs.h) is a *Grid_Map* with cells of type *Goodness_Cell* and a member variable for minimum certainty of the map. There are getters for height, certainty, and goodness of a cell. There are functions to: find the cell with the worst combination of certainty and goodness; generate an *RGB_Image* showing cell height or cell goodness (see *Goodness_Cell*); merge the data from a second map; and do stream and file I/O. The class takes an *Indexer* as a template type. **Goodness_Map** and **Scrolling_Goodness**_Map are *Goodness_Map_Templates* with the standard and wrap around Indexers.

- **Cspace_traversability** (analysis_terrain_morphin – morphin_analysis.h) has a bunch of parameters to collectively describe the *Grid_Map* cells that a robot of known footprint would cover when sitting over a particular cell.

- **Morphin** (analysis_terrain_morphin – morphin_analysis.h) calculates a *Goodness_Map*. Function Compute_goodness_map() loops through each cell in a *Plane_Fit_Map,* computes a *cscape_traversability,* and repackages it as a *Goodness_Map*. Helper function compute_cell_traversability() computes the *cscape_traversability*. It moves a (parameter) robot region (i.e. footprint) to a (parameter) location and orientation on a (parameter) *Plane_Fit_Map*, records all cells under the region, and accumulates them into a *Plane_Fit_Moment*. It calculates the worst residual and variance of either a set of cells around the robot center or of the accumulated cells under the robot region minus any one cell, neither of which make much sense to me. The constructor copies a *morphin_analysis_params*, which the class uses frequently.

- **Morphin_Analysis_Params** (analysis_terrain_morphin – plane_fit_map.h) is a set of parameters for running morphin. Has a *config_module*, which we don't yet describe, and many threshold values and switches. Constructor sets default values for thresholds and switches and adds each variable and a description to the *config_module*. There are pass-throughs to the *config_module*'s file I/O functions. The constructor can use these to read from a file at an optional, parameter filename. There are a couple of setters and a function to copy many parameters from a *Wheel_Locomotor_Model*.

4.6 Stereo

Stereo is organized as a base class that defines a common interface and a number of child classes that define specific stereo implementations.

- **Stereo_Processor_Impl** (stereo_processor – stereo_processor_impl.h) is a base class for stereo implementations. It records a pair of *Camera_Images* (inputs); a disparity map, a range map, and a point cloud (outputs); and min and max disparity, correlation window dimensions, and whether to filter the stereo results (parameters). It has setters for the inputs and parameters, getters for the outputs, functions to calculate each of the outputs, and flags to say which inputs and outputs are actually filled. The constructor sets parameter defaults and can copy an optional pair of *Camera_Images*. Child classes must define calc_disparity() and can define calc_range() and calc_point_cloud() or use default functions that do nothing.

  Stereo_Processor (stereo_processor – stereo_processor.h) is an apparently aborted precursor to *Stereo_Processor_Impl*. It declares functions to set/get a pair of *Camera_Images*, get (and presumably calculate) disparity and range maps, and set window size, max disparity, and post-filter-enable. It does not define the functions or member variables.

- **Stereo_ Impl_SVS** (stereo_vision_svs – stereo_impl_svs.h) is a child of class *Stereo_Processor_Impl* that wraps SRI's SVS stereo. It requires header files and libraries from the SVS package. These are not CLARAty but are at paths specified in the module's Makefile. The class defines the functions declared in the *Stereo_Processor_Impl* interface and adds a file reader and getters and setters for confidence threshold and horopter X offset. Most of these functions access two new member variables – a stereo operator object and a stereo parameter struct. The class defines calc_disparity() but only has stubs to calculate range map and point cloud.

- **Stereo_Impl_Jpl** (stereo_vision_jpl – stereo_impl_jpl.h) and **JPLStereo** (stereo_vision_jpl – JPLStereo.h) are a child class of *Stereo_Processor_Impl* that wraps JPL's stereo algorithm, and the actual code for that algorithm. They will probably change shortly, so I'm not documenting them yet.

- **Stereo_Impl_WBS** (stereo_vision_wbs – stereo_vision_wbs_impl.h) is apparently Clark Olson's wide baseline stereo. It deserves to be documented here, but it is also likely to change soon.

- **Disparity_Correlator** (arc_vision – disparity_correlator.h) calculates disparity and mask images from a stereo image pair. Apply(), does 2D correlation, left/right consistency check, outlier removal, and (optional) hole filling. The correlation step determines whether the input images are binary, and then uses one of two correlators, for binary or regular images. Correlators use sum of absolute difference, implemented as an exclusive-or, which I don't really follow, but which probably makes sense for compressed, binary images after the *filter_slog* operator. Outlier removal masks out disparity pixels whose values differ by more than a threshold amount from at least 60% of the pixels in the local 11x11 window. That is probably faster than blob filtering, but will eliminate thin objects. Hole filling fills masked-out pixels by scanning left and right along the scan line to the first non-masked pixel in either direction, and accepting the larger disparity of the two. Setter functions let you specify min and max horizontal and vertical disparity, correlation kernel size, and outlier threshold. You can also specify whether to find horizontal and/or vertical sub-pixel disparity (parabola fit) and whether to fill holes.

4.7 Other Algorithms

- **Nelder_Mead_Minimizer** (Numerics – nelder_mead.h) implements the Nelder-Mead Simplex Algorithm, which minimizes a user-supplied, multi-variable function. The algorithm is rumored to be inefficient but capable of handling non-continuous functions, without using derivatives, and without being sensitive to starting values. Two equivalent member functions, nelderMeadMinimize() and minimize(), take a function to minimize, an initial guess at the minimum values, a maximum number of iterations, and the maximum allowable error. They overwrite the initial values with final values. NelderMeadMinimize() takes a target function that reads an array of doubles and returns a double, while minimize() uses templates to take any function whose output can be cast to a double and whose arguments can be cast from an array of doubles.

- **Numeric_Solver_1D** (solver_1d – numeric_solver_1d.h) is a class to find a root or minimum of a 1D function. Function find_root() does Newton minimization to find a root near the initial guess. Use constructor or setters to specify initial guess, cost function (*Function_1D* pointer or a function that takes and returns a double), perturbation (step size for calculating derivative), max number of iterations, and max acceptable cost. Minimize() finds the (presumably non-zero) minimum of a cost function. It takes as parameters three 1D points, in order, and their cost values. It applies a parameter number "bisections", each moving the first and third point in toward the second and adjusting the position of the second to move all three toward a minimum. If finally returns the second point.

  Function_1D (solver_1d – function_1d.h) is a base class with a single virtual function (must be redefined) that both takes and returns a double. It is used to store the cost evaluation function for *Numeric_Solver_1D*, though it is not clear why this should be preferred over a global function that takes and returns a double.

- **ransac()** (arc_vision – ransac.h) is a standalone function that implements the RANSAC (random sample consensus) algorithm for finding the transform that aligns a (parameter) point cloud with a (parameter) model. It iterates three steps: sample the input points; determine the transform to best fit them to the model; and count the number of input points that are within (parameter) threshold distance of the model according to a (parameter) test. Iteration stops after a (parameter) maximum number of iterations or if a (parameter) fraction of the input points are within threshold distance of the model. The model parameter is of templated type. Its class must have: a "model" field, which stores the transform from points to model; a function get_minimum_sample_size(), which returns the number of points required to estimate the transform; and an operator() that takes an array of points and an array of indices for those points, estimates the transform from those points to the model, and stores the result in the "model" field. The class of the point set parameter must include a size() function that returns the number of points. The algorithm's sampling step does not check whether any of its samples are duplicates.

- **FindSun** (JPLPic – jplpic.h) is a *JPLPic* member function that finds the centroid of the sun in a *JPLPic*. The algorithms is: convert image to 8-bit; calculate 0.9 * maximum image intensity and the intensity of the top 5% of the image histogram; threshold the image at the larger of the two; blob find to locate high intensity areas; and find the centroid of the largest blob. The function does not use existing *JPLPic* functions to histogram, find top 5% threshold, or threshold the image. It does not calculate the largest blob centroid during blob marking, which would be faster. It probably belongs as its own operator, not as a member of JPLPic.

- **Msky()** (analysis_edge – sky.cc) finds a horizon in an image by looking at the pixels. Divides image into vertical swaths. Starts a marker at the top of each swath. Lets each marker drop until it reaches the nominal horizon based on camera elevation & roll (or image bottom if no camera info), or until it drops 5 pixels below its neighbor on either side, or until the variance in a window around the marker exceeds a (parameter) threshold. The process repeats as long as at least one marker drops and then stops due to the variance threshold. If no markers reach the variance threshold, the threshold is reduced to the highest variance among the markers, and the process repeats. If no markers drop, the function calculates a salience metric that considers the fraction of markers that stopped due to high variance and the sum of current marker variances versus the average variance of pixels above the markers. If the salience metric is the highest yet encountered, the function stores the positions of the markers as the best horizon, and then it raises the threshold and continues iteration, allowing swaths that had previously hit a threshold to continue dropping. This process continues until the variance threshold exceeds another parameter or all markers drop to their lower limit, at which point the function returns the best horizon. If nominal horizon goes above image or variance at top box is too high, the function aborts. There is at least one error in loop dims using BOXWID vs. BOXLEN.

- **Specpoint()** (analysis_edge – specipoint.cc) extracts targets from a list of edges. You provide a source image, the corresponding images of edge magnitude and orientation, a linked list of *Edges* (each a list of pixels), the camera angles, the sun angles, an FOV (in pixels), and a confidence threshold. The function zeroes edge magnitudes at image border and where edge orientation is more than 90 degrees from angle to sun. Next it places a one-FOV disk for each edge, centered halfway along the segment between edge endpoints and then moved just over ½ FOV normal to the segment, toward the sun. Next it evaluates confidence for each edge, with a metric that increases with the mean intensity in the disk, decreases with variance in the disk, and increases as the average intensity (using only the low 70% of histogram) of pixels near the edge pixels decreases. Finally, it fills a parameter *Targets* with FOV center, confidence, and size (distance between endpoints) of up to 100 edges whose confidence exceeds the parameter confidence threshold, starting with the smallest edge. There is commented-out functionality to mask out pixels above the horizon.

- **Linlayers()** (analysis_edge – linlayers.cc) takes an *Edge* list (a list of lists of connected pixels) and generates an array of three images showing the locations of large numbers of similarly oriented edge pixels. Conceptually, it has three steps. First, it generates a binary image showing all edge points that are not within a parameter *length* of the ends of edges. Second, it passes a window of (parameter) *side-length* across that image, creating for each window a histogram of the "orientations" of edge pixels within the window. Number of histogram buckets is a parameter. The orientation of point P on an edge is the angle (CCW from x axis) of the line between points P+*length* and P-*length* on the same edge (hence points within *length* of edge ends are ignored.) Third, the function creates and returns an array of three images. The first gives the fraction of the window around each pixel that is covered by edge points, scaled from 0 (0%) to 255 (5% or more.) The second gives the fraction of the edge points in each window whose orientations are most populated, neighboring pair of histogram buckets. This is scaled 0 to 255, showing the agreement between orientations within each window. The third is a thresheld version of the second, showing 255 where "layers" produce consistent orientations and 0 where arbitrary edges do not. Here are some picky notes. The input window size is even or rounded down to an even number for the second pass. The output images are shifted up and left by ½ window-size from the input edge coords, and a 1-window-size border on the right and bottom of the images are not processed. Edge pixel coordinates are given as (x,y) with y=0 at image bottom. These are translated into (r,c) with r=0 at top to create the initial edge image. Windows with more than 255 edge pixels are not handled precisely. The third (thresheld) image also requires that the fraction of a window containing edge pixels exceed 0.3% of the area of the window, which is probably a bug that intended to verify that edges occupy some minimum fraction of the window.

- **LayerAnalysis()** (analysis_edge – layeranalysis.cc) makes a list of ellipses representing blobs in a binary image. It takes two concentric binary images, one being smaller by (window_size-1)/2 for (presumably odd) parameter window_size. The function extracts all regions of 4-neighbor connected pixels where both images have "on" pixels. It throws away any regions whose size (number of pixels) does not exceed (parameter) min_size. For the largest (parameter) maxblobs regions, it calls CreateEllipse() to generate the best fit ellipse for the region. If the larger image (presumably a mask) is NULL or the wrong size, it is assumed to have all "on" pixels. If maxblobs exceeds hardcoded MAXBLOBS, it is reduced to the latter. It fills parameter number of actual blobs and returns the list of ellipses. It turns off all of the pixels of the smaller image.

- **Science_Analysis** (Analysis_edge – Science_analysis.h) is a class with 6 functions that are just wrappers to call canny.filter + rescale.filter, msky(), linlayers(), specpoint(), layeranalysis(), and Cross_hair(). The first 5 do various image interpretation tasks, while Cross_hair() does image markup.

- **CheckBorders()** and **_CheckBorders()** (analysis_edge – sky.cc) find the leftmost, topmost, rightmost, and bottommost row/column of a byte image for which the row/column average exceeds 1. For CheckBorders(), acceptable row/column must also have variance within some threshold of that of the next-inner row/column. Neither function is actually ever used.

- **FOVMeanStd()** (analysis_edge – specipoint.cc) calculates the mean and deviation of intensity (0-1) of pixels in a disk, given center and radius.

- **SeaLevelHorizon()** (analysis_edge – sky.cc) calculates the 2D, in-image-plane line (x0,y0,m) of the sea-level horizon, given camera elevation, roll about center pixel, and angle/pixel (does not interpret image). It can optionally draw a black line at the horizon.

- **Geom2FilterDir()** (analysis_edge – specipoint.cc) returns angle to sun in image plane given sun az/el and camera az/el/roll in world coords.

- **NormShadowValue()** (analysis_edge – specipoint.cc) takes a list of edge pixels, creates a line segment at each (from –0.5σ to 1.5σ for param σ, oriented according to a parameter orientation image), histograms the points in the segments, and returns the average intensity (0 to 1) of the lower 70% of the histogram. Variable hist[] is allocated 1 entry low. Ignores σ/2 points on either end of the edges list.

4.8 Non-vision functions you might use anyway

- **String handling functions** (share – string_util.h). Functions to copy a string and strip specified characters from front and/or end. Functions to check whether a string has a specified character, and to remove or replace everything after its last occurrence. Structures that have these first two sets of functions as operator(), as in functors, below. Functions to extract the first double, long, bool, quoted string, or token (specified characters or delimiter), or to extract all tokens, from a string. Function to put quotes around a string and convert any existing quotes into backslash-quote. Define string_printf that writes a string, unless vxworks.

- **More string handling functions** (jplpic – Mwm.h, cistr.c, filename.c). Case-insensitive string comparisons **cistrcmp**() and **cistrncmp**() are not elegant, but could be added to the Share module's string_util.h. **Mklow**() wraps tolower. Cistrcmp() is defined only in cistr.c, but is unused. Cistrncmp is defined in both cistr.c and filename.c, while mklow is defined in both mwm.h and filename.h, suggesting that cistr.c is unnecessary. Mwm.h has a lot more function declarations, macros, and structures, but virtually none of it is used.

- **hex_to_int()** (share – common_defs.h) extracts the integer value of a hex number (starts with "0x", contains only 0-9 and a-f, ends in whitespace, \n, or \0) embedded in a string. It cannot handle digits A-F, dies ungracefully if an illegal character is in the number, and does not recognize \t as ending the number. If you make a better routine, you should offer it to the repository.

- **Operators << and >>** (share – string_defs.h) are redefined for the templated *basic_string* type, for a version of vxworks that had defined them in a cc file where the templates would not be properly resolved.

- **order()** (share – common_defs.h) returns the order of magnitude of a float.

- **cl_sleep()** (share – common_defs.h) sleeps for a parameter number of seconds if you are using C++ and ACE. Does nothing otherwise.

- **Byte parsing** (share – common_defs.h) defines macros MSB, LSB, MSW, LSW to return top or bottom byte/word of a short/long.

- **Math helper functions.** (share – common_defs.h) has C++ macros cl_min (2 args), cl_max (2 args), cl_min3, cl_max3, cl_sgn, cl_sqr, cl_abs, cl_in_range (max > x > min). It also defines sgn and sqr for legacy vxworks code.

  There are redundant definitions for many of these (JPLPic – real_helpers.h), including an absolute value function that will probably fail on very small numbers and may account for why other *JPLPic* files redefine ABS. The latter file also defines DEG2RAD and RAD2DEG macros. It defines an epsilon and macros to test two numbers: GT/LT than by more than epsilon, GE/LE/EQ/IN with allowable error of epsilon, where IN means between two values. There are also EQe (provide your own epsilon) and INe (provide your own, larger epsilon). A second set of comparisons decides whether the difference between two numbers is approximately GT/GT/LT/LE/EQ/EQe a multiple of a parameter period. There are functions to shift an angle (in radians) to the ranges 0 to 2pi or –pi to pi, though the use of the approximate comparisons makes them a little sloppy at the ends of the range. There are functions to find the smallest signed or unsigned distance between two points on a number line that wraps around at a parameter period. A function maps a float number from a float range to an int number line. A function rounds double to int with symmetry about 0. Macros do fmod (float modulus) but return a positive remainder or round very high remainders to 0. (JPLPic – mwm.h) defines (unused) in_range(), int_min(), and int_max() that are redundant with cl_* versions.

- **Fop(a,b)** (analysis_edge – specipoint.cc) is true iff a, b, and a>=b.

- **Error handling** (share – common_defs.h) defines several functions that logmsg and exit(1) to report various errors such as divide-by-zero, so that you don't have to write your own printfs. Also, if c++, defines macros to delete and null pointers, write a message plus current __LINE__ and __FILE__ variables (whatever those are), and time a code block. Also, (JPLPic – real_helpers.h) has believe*(), which replace assert() by returning 0, INTERNAL_ERR, void, or parameter value if a parameter test fails.

- **More error handling** (jplpic – ErrHandle.h). A bunch of macros, consts, and functions for reporting errors. Has enum types NavErr (error return values) and ErrorSeverity. Bunch of #defines and #undefs if MER. #defines BOUND() to abort or ignore on memory overrun. #defines true and false, which could be an issue for C++, and probably is redundant with common_defs anyway. Defines Warning(), Message(), FatalErr(), error(), err_print(), DGB(), and VDBG(), all of which are basically printf. #Defines ERR. Defines a bunch of EINFO*, EWARN*, and EFATAL* macros that generally resolve to printf, and are likely used for debugging. I imagine that everything but NavErr may belong in a CLARAty include file, while NavErr can move to nav_memory.h.

- Display.h (share – display.h) has several functions to format screen output of text.

- **Arithmetic and logic member functions** (share – various files). CLARAty provides a short cut to add arithmetic and logic operations to a class definition. Claraty_gendefs.h defines 2-byte bit flags (CL_OP_*) to represent standard arithmetic and logic operators. If you #include claraty_gendefs.h, #define CL_OP_MODE to some combination of these bit flags, #define CL_OP_TYPE to the name of a regular class or CL_OP_TEMPLATE to the name of a templated class, and then #include claraty_opgen.h (formerly operators.h), then the compiler will define class members for the operators you specified. You can #define other CL_OP_* constants to handle cases such as classes with multiple template args or defining members for specific template instantiations. I'm not sure how useful claraty_opgen is, because its member functions rely on your class already having definitions for ==, +=, etc. The only files that actually use it are the Matrices module's matrix_operators_old.h and the bits module's bits.h. If you #include claraty_gendefs.h, #define CL_OP_NAME to a class name (including template specifier if the class is templated), #define CL_OP_MODE as above, and #include claraty_expgen.h, then the compiler will declare headers for the operators for the class. This is a good way to instantiate templates with specific template types. Only the Matrices module's matrix_templates.cc uses this.

- **Cl_*() List processing functions** (share – claraty_functors.h). These functions process input list(s) to modify and return a scalar. The list(s) and scalar are inputs with different template types. The first list is specified by first and last element (pointers or iterators.) The input scalar establishes the output type. Presumably it would be good to add things like mean and variance to this set.

    - **cl_inner_product()** returns the dot product of two input lists, added to the scalar input.

    - **cl_accumulate()** returns the sum of the elements on a list, added to the scalar input.
    - **Cl_reduce()** takes a list, a scalar, and a binary cl_* operator (see below), walks down the list applying the formula "scalar = op(scalar, *list++)", and returns the modified scalar.

- **Applying a function to each member of a list** (share – claraty_functors.h).
    - **cl_* operators** are templated classes with operator() defined. They include cl_round (to nearest int); cl_left_shift and cl_right_shift; cl_bit_and, cl_bit_or, cl_bit_xor, cl_bit_not; cl_lessthan_op, cl_lessthan_equal_op, cl_greaterthan_op, cl_greaterthan_equal_op (compare); cl_rms_op (hypotenuse); cl_sumsq_op (hypotenuse squared); cl_max_op, cl_min_op, cl_plus, cl_minus, cl_multiplies, cl_divides; cl_abs_op. Operator() has one or two inputs and one output, all of which share one data type except for the four comparison operators (bool output), cl_round (specify output type too), and cl_*_shift (number of bits to shift is a separate type.) You instantiate one of these structs with no constructor args, as an argument to *cl_transform*(), *cl_apply*(), or *cl_bind2nd*(). Why not use functions instead? Perhaps because you can't instantiate a templated function as an argument to another function, and not all classes have already-instantiated operators such as operator+ defined, and we want to be consistent.
    - **cl_transform()** applies a unary or binary cl_* operator to one or two lists of inputs to produce one list of outputs. You provide the first and last element of the first list (presumably pointers or iterators), the first element of the (optional) second input list, and the first element of the output list, all of which are templated types. **Cl_apply()** wraps *cl_transform()* and applies unary, cl_* operator, in place, to a list (any stl container type).
    - **cl_bind2nd()** makes a new unary cl_* operator from a binary cl_* operator and an argument to be used as the second operand of the cl_* operator. **Cl_bind()** makes a no-argument cl_* operator from a unary cl_* and an operand. Typically, you would send the result to *cl_apply*() or *cl_transform*() to apply an operator to each element of a list, using the same operand. The new operators are *cl_binder2d* or *cl_binder* structs.
    - **cl_simple_function**, **cl_unary_function**, and **cl_binary_function** are the base classes for cl_*. They just define typedefs for the class' templated types. I do not know what value they add. A common cl_* base class with virtual operator() would let you declare cl_transform() with the base class as a parameter, but we don't do that. Only cl_unary_function appears outside claraty_functors.h, inherited by functions that check if a point is within a bounding box – a good place for a cl_transform.
    - **Vfunctor*** are three classes of templated type, whose operator() takes 0-2 args and returns a new instance (no constructor args) of the first templated type. They appear only in obscure, non-vision code.
- TIME() (camera_model_jpl – Timing.h) is a macro that returns either the current time or the time since the previous call.

5. Constants

Here are some constants that are used in the various CLARAty modules.

- **Standard CLARAty Constants** (share – common_defs.h). This must be included before iostream.h.
    - If vxworks: define NULL if needed; include some vxworks libs; define SHORTSWAP depending on _BYTE_ORDER.
    - If vc++, include iostream and suppress some compiler warnings. I believe that was necessary when I used vc++, but I don't think it worked. So the test for whether you are vc++ may have failed, or the version may have improved since then.
    - If not vxworks: define logmsg and taskdelay
    - Defines BYTE, WORD, DWORD, ON, OFF, YES, NO, OK, ERROR, POSITIVE, NEGATIVE, ABSOLUTE, RELATIVE
    - Includes math_constants and unit_conversions.
    - Defines some DEBUG switches, CLEAN, and some KEYBD key codes
    - Defines MARK_UNUSED, which can somehow be used to keep a compiler from whining that a variable is not used.

  If not c++, defines bool as int.

- **Unused JPLPic constants** (jplpic – viscommon.h) are TRUE, FALSE, SUCCESS, FAILURE. Nobody uses them, but some files define and use constants of the same name.

- **Redundant jplpic constants** (jpl_pic – real_helpers.h) include M_PI, which is already in the Share modules's math_constants.h and several macros that are already in the Share module's common_defs.h.

- **Unit conversions** (share – unit_conversions.h). CL_* macros convert between lb/kg/N, sec/msec/usec, m/km/cm/in, deg/rad, sin/cos. File includes math.h and is included by share's common_defs.h.

- **Math constants** (share – math_constants.h) include common manipulations of e, pi, root2, and epsilon (very small number.) Included by share's common_defs.h.

Appendix: Modules covered in this document

The following modules from the module database (as of 3/30/04) looked vision related. Blue modules are factored into the classes/functions above. Grey modules look relevant but had no readme files or had only the default sample files, so they were not yet real, and they are not described above. Purple modules are not factored in above because they are going to change soon. Black modules are not factored in above because I just didn't get to them. You can find newer modules and releases by looking at the CLARAty website under software, packages, which gives the complete module data base, listed in order of most recent release.

Arc_vision-r1-00a
Analysis_carbonate-R1-01 (see note 2)
Analysis_ellipse_detect-R1-00
Analysis_region-R1-00
Analysis_rock_finder_oasis-R1-00b or
analysis_oasis_rockfinder-R1-00
Analysis_spectra_bayes-R1-00
Analysis_shape_detection-R1-00
Analysis_terrain_morphin-R1-02a.
Analysis_vista-R1-00
Analysis_edge-R1-00a
Arc_slog_tracker-r1-00a
Arrays-r1-08b

Camera-r1-03c
Camera_image-r1-04a-build01
Camera_image_io-r1-00c
Camera_image_io_pds-r1-00
Camera_model-r1-04a-build01
Camera_model_jpl-r1-01c

Corner_detect_op-R1-01
Crater_detector

Data_io-r1-00d
Diag-r1-00
Draw_ops-r1-02

Edge_detect_op-R1-01

Feature_tracker-r1-01
Frame-r1-06b-build01
Frame_tree-r1-00
Fuzzy_logic-r1-00
Fuzzy_logic_utils-r1-00

Image-r1-04c-build01
Image_displayer-r1-03-build01
Image_ops-r1-04a
Image_pyramid-r1-01
Image_rgb-r1-01a

Image_io-r1-03d
Image_io_pnm-r1-01a
Image_io_tiff-r1-03 and
Image_tiff_io-r1-01
Image_monitor-r1-00

Jplpic-r1-01c
Jplpic_file_io-r1-00
Jplpic_libmwm-r1-00

Localizer_visual_olsen-r1-00

Map_grid-r1-01b
Matrices-R1-09c
Matrix_n_exp_n-r1-00

Numerics-r1-01a-build01

Parameter_parser-r1-00
Points-r1-08a
Point_cloud-r1-02a
Point_image-r1-01a
Pose_estimator_ekf_6d-r1-00
Project_2d3d_tracking
Project_camera_group_1394-R1-00

Qt_camera-r1-00
Qt_image-r1-01a

Rectify_op-r1-03a
R8_camera_models-r1-00

Science_data-r1-00
Share-r1-09a:
Solver_1d-R1-03b
Stereo_processor-r1-03b
Stereo_vision_jpl-r1-04d
Stereo_vision_wbs-r1-00c
Stereo_vision_svs-r1-03

String_io-r1-02c
Transforms-07b-build01
Tree-r1-00

Util_open_gl-r1-01a
User_rich

Visual_odometry-r1-01b
Visual_odometry_jpl-r1-02a
Visual_odometry_tester-r1-00b
Visual_tracker-r1-00

Note 1. I have not investigated the navigator and navigator_morphin classes, which did not seem sufficiently vision related. I have not investigated specific Camera_*Impl_** classes, which included the following: Camera_pxc200-r1-01c, camera_px610-r1-01, camera_vx1394-r1-02-build01, camera_ieee1394-r1-01a, fd_camera-r1-01a-Build01, camera_linux1394-r1-03b, camera_v41-r1-02, and fd_camera. You get the idea of *Camera_Impl_** from the generic text above.

Note 2. Analysis_carbonate is not a vision-related module. It seems to operate on spectrum data instead. It defines the *Carbonate_identifier* and *Nested_list* classes the former is complicated and not vision relevant, so I won't describe it here. The latter is not vision related, but I already spent the time to describe it, so here is the description. **Nested_list** (analysis_carbonate – carbonate_identifier.h) is a linked list node that can be a float, int, string, or array of *Nested_list*. It has a variable for each type, a flag to say which is active, and a next pointer. It has getters for all of those. Functions to modify the list: init, free, assign, copy (single element or also its list), insert, append, find, remove, compare, read/load (two functions, somehow different), and write. Constructor can copy another *Nested_list* or a string representing a *Nested_list*. Not sure how this class does/should relate to *FDM_Node* (see appendix.) Not sure why the various copy commands and constructors are separate. Not sure whether there is already an STL class that handles nested lists.

Appendix: The Data_IO Module and FDM Class

The Data_IO module facilitates reading/writing new classes to/from "ACE CDR" and "Parse_Block" (perhaps the same as Parse_Tree) formats. The functionality is not computer vision related, but it appears in several vision modules, so you might be interested.

The Part You Might Care About

The Data_IO module defines two important base classes: *FDM_Node* and *FDM_Stream*. *FDM_Stream* provides a common format to store data from an arbitrary class along with I/O functions for that data. Sub-classes of *FDM_Stream* read/write specific file formats – currently ACE CDRs and Parse_Trees. *FDM_Node* has an *FDM_Stream* and not much else. Sub-classes of *FDM_Node* generally represent arrays or maps of *FDM_Nodes*.

To use Data_IO, an arbitrary class (we'll call it *myclass*) must define a function to translate its member data to/from the *FDM_Stream* member of a sub-class of *FDM_Node* (we'll call it *subclass*.) The function can be either, "bool *myclass*::**io** (*subclass*)," or, "bool **io_object** (*subclass*, *myclass*)." *FDM_Stream* provides helper functions to use in io() and io_object(), as well as functions that use io() or io_object() to convert the whole block of data between *FDM_Stream* format, a *myclass* object, a string, a stream, a file, a *Parse_Tree*, or an *ACE_Message_Block*. So *myclass* needs only the one new function in order to get all of that functionality.

For usage examples, grep for io() and io_object() in the repository or see
http://claraty.jpl.nasa.gov/new_site/project/meetings/2003-04-10/fdm_2003_04_10.pdf .

Basic Types

Parse_Tree and Parse_Tree_Data are used to build a tree whose leaves are strings. Unlike your standard tree or linked list, each type points to nodes of the other type, so node types alternate as you proceed down a branch of the tree.

- **Parse_Tree_Data** (Data_io – parse_tree.h) can represent a string, an STL vector of *Parse_Tree*s, an STL map of strings-to-*Parse_Tree*s, or null. It has a flag to say which type of data is represented, three variables to store data for the non-null data types, and a setter to set the flag and clear the unused variables. It also has a *refcount* and functions to increment/decrement/destruct it.

- **Parse_Tree** (Data_io – parse_tree.h) has a *parse_tree_data* pointer and functions to operate on it, including: getters/setters for pointer's data type and map/array elements; getters for the pointer's map and map/array size; test of whether the pointer's map has a (parameter) label; copy from arbitrary object (converting to *parse_tree_data* via a FDM_parse_tree); and string/stream read/write. I/O format is { label=value; … }for maps, [element …] for arrays, strings quoted if they have confusing characters, and multi-line (e.g., nested) maps/arrays indented.

Construct a *parse_tree* empty, or copy another *parse_tree* or a string representing a *parse_tree*. Operator= and destructor handle the data pointer's *refcount*.

FDM_Stream and its sub-classes

- **FDM** (data_io – fdm.h) defines an enum with values Read and Write. It is used to describe the direction of an *FDM_Stream* and its descendents.

- **FDM_Stream** (data_io – fdm.h) is a base class with four main parts. First, it has an read/write flag, which is copied from a parameter *FDM* in the constructor and accessed by getters. Second, the class has a vector (stack) of "nodeinfo_strs", each having an id (of an *FDM_Node*) and a *refcount*. A function to push an *FDM_Node* onto the stack actually creates/pushes a nodinfo_strs node, assigns it the next available id and *refcount*=1, and records the id and stack depth in the *FDM_Node*. There are also functions to manipulate and search the stack. Third, the class declares many virtual functions that must be overridden by descendents. These include io_object() for standard data types; wrapped io_object() calls for strings and ints; and functions to manipulate the *FDM_Node* corresponding to the top element of the stack by popping it, setting its type (array or map), getting the length of its array, copying a map field or array element to/from a new node and pushing that node onto the stack. Fourth, the class defines some functions that rely on virtual functions, thus setting an interface but not its functionality. One is a member io_object() that takes a templated parameter and calls the global io_object() defined with that parameter's class, or if that class has no io_object(), calls a global **io_object**() that calls the parameter class' io(). Of course, the specifically-declared, virtual io_object() members have precedence over the templated one. Two other functions read and write by wrapping an io_object() call with calls to member functions that do nothing by default.

- **FDM_Parse_Tree** (data_io – fdm_parse_tree.h) is a subclass of *FDM_Stream* specialized to read/write a *Parse_Tree*. It overrides the *FDM_Stream's* stack with a vector (stack) of *_Stack_Elt*, each containing a parse tree (the important part) and an index (used if the parse tree is an array). Its constructor can copy this stack from a parameter *Parse_Tree.* The class redefines *FDM_Stream's* virtual functions for stack handling. Most io_object() functions are defined to call member function io_primitive(), which gets/sets an arbitrary object from/to a string stored in the parse tree at the top of the stack. The module defines static functions that invoke an *FDM_Parse_Tree* to convert an arbitrary object to and from a string, stream, or *Parse_Tree.*

- **FDM_Parse_Tree_File** (data_io – fdm_parse_tree_file.h) is a sub class of *FDM_Parse_Tree* specialized for writing to files. It has functions to open/close an fstream to a file, to read/write a 4-byte "magic number", to read/write an arbitrary *Parse_Tree* or the *Parse_Tree* at the top of the stack, and to read a *parse_tree* and print a summary of its contents. Read/write direction is set in constructor. There are several flags to make sure I/O functions are called in a reasonable order. The module provides static functions to read/write an arbitrary object from/to a file via an FDM_Parse_Tree_File.

- **FDM_CDR** (data_io – fdm_cdr.h) is an FDM_stream. It is about the same as *FDM_Parse_Tree*, except it operates on *ACE_Message_Blocks* instead of *Parse Trees*, and it defines the << and >> operators.

- **FDM_CDR_File** (data_io – fdm_cdr_file.h) is a subclass of *FDM_CDR* that is specialized to read/write files. It is basically the same as *FDM_Parse_Tree_File*, except that it uses ACE instead of normal file handling, and presumably it writes CDR things instead of *Parse_Trees*. It has static functions to convert between file and message block, static function to show file contents, and a pass-through to FDM_CDR's check_streams().

FDM_Node and its sub-classes

- **FDM_Node** (data_io – fdm.h) has an *FDM_stream*, integer id and level (used by the stream), and constructors. All are protected, so you must inherit the class.

- **FDM_Map** (data_io – fdm.h) is an *FDM_Node* specialized to process data stored as label-value pairs. Its constructor copies an *FDM_Node* and pushes the copy onto its own *FDM_Stream's* stack. The class has a function to locate itself of its *FDM_Stream's* stack, extract a value for a given label from the stack into a new node, push that onto the stack, and read/write it to/from a parameter object.

- **FDM_Array** (data_io – fdm.h) is an *FDM_Node* with some extra functions. Specifically, there is a function to determine the length of the array stored in the node's stream, and functions to copy the "next" element(s) from that array to the top of the stream's stack and then to/from a parameter object. Can construct by copying another *FDM_Array* or by copying an *FDM_Node* and stacking the copy in the new node's stream.

- **FDM_Untyped_Node** (data_io – fdm.h) is an *FDM_Node* with a function, value(), that converts between the *FDM_Node's* stream and a parameter object using the object's io_object() or io() member.

- **FDM_Singleton** (data_io – fdm.h) is an *FDM_Node* with a pass-through to an undefined object. Construct from an *FDM_Node* or a *FDM_Singleton*, though the latter is not defined.

Other Stuff

There are some additional pieces that seem irrelevant, but since they were there, I have documented them.

- **fdm_error** (data_io – fdm.h) couts a char * with some text on either side.

- **FDM_File** (data_io – fdm_file.h) has static functions to open a stream to a file and to open, read, summarize, and close a file. Both read from an ACE socket to determine whether the file is CDR or *Parse_Tree*, and then call *FDM_CDR_File* or *FDM_Parse_Tree_*File functions.

- **ACE_transfer_helper** (data_io – ace_transfer.h and .cc) has a single, static function to send a large *ACE_Message_Block* (data packet) over an *ACE_SOCK_Stream*.

- **Data_Receiver** (data_io – ace_transfer.h and data_sender.def.h) is an ACE socket, an address, and an is-connected flag. It has functions to connect to a host and port, read data (templated type) from socket, and disconnect.

- **Data_Sender** (data_io – ace_transfer.h and data_sender.def.h) has an address to listen to, a flag to record whether it is listening, and a (templated type) object from which it can read data. The constructor records the data-reading object. Has a function that, through a surprisingly complicated series of other functions and spawned threads, listens at a parameter port, accepts a connection, calls the data-reading object to get some data, and sends it on the connection (possibly using *ACE_Transfer_Helper*.)

- **Data_Publisher** (data_io – ace_transfer.h and data_sender.def.h) lets you send data to multiple places. Constructor creates and registers a member event handler. Start() spawns a thread that accepts ACE socket connections and adds them to a member subscriber list. Stop() kills the thread and closes any subscribed sockets. Publish() pushes (templated type) data out along all subscribed connections.

- **Data_Subscriber** (data_io – ace_transfer.h and data_sender.def.h). Subscribe() connects to a *Data_publisher* at parameter host and port, then spawns a thread that continuously tries to read from there into a member buffer. Two functions copy the buffer to a parameter buffer, differing by some use of a mutex. Unsubscribe() shuts down the thread.